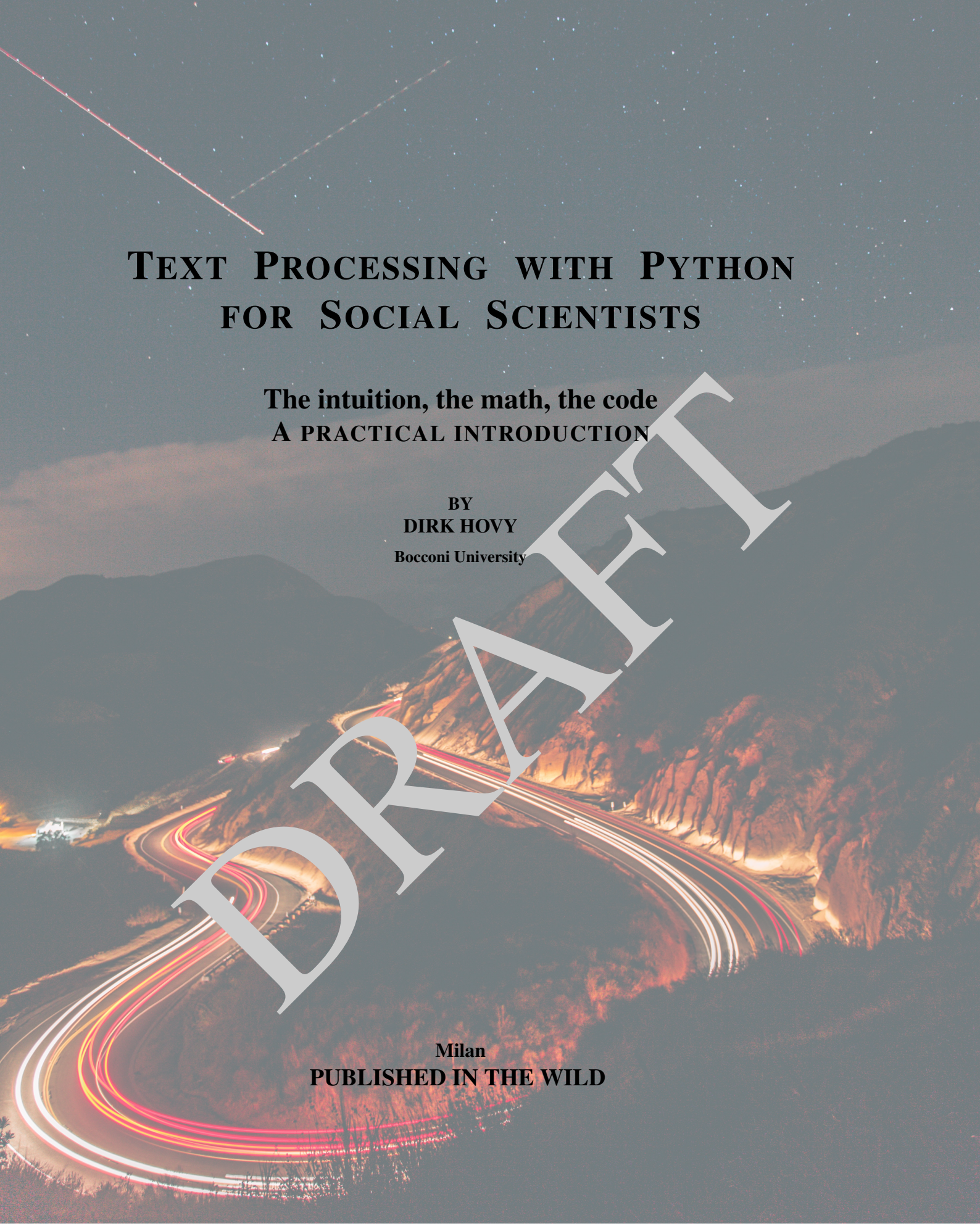


TEXT PROCESSING WITH PYTHON FOR SOCIAL SCIENTISTS

The intuition, the math, the code
A PRACTICAL INTRODUCTION

BY
DIRK HOVY
Bocconi University

Milan
PUBLISHED IN THE WILD

Contents

Introduction

Today, text is an integral part of our lives, and one of the most abundant sources of information. On an average day, we read about 9000 words. That includes emails, text messages, the news, blog posts, reports, tweets, and things like street names and advertisements. Throughout a lifetime of reading, this gets us to about 200,000,000 words. It sounds impressive (and it is), and yet, we can store this amount of information can in less than half a gigabyte: we could carry a life's worth of reading around on a USB stick. At the time of writing this, the Internet contains an estimated minimum of over 1200 TB of text, or 2.5 million lives worth of reading. A large part of that text is now in the form of social media: microblogs, tweets, Facebook statuses, Instagram posts, online reviews, LinkedIn profiles, YouTube comments, and many others. However, text is abundant even offline – in quarterly earnings reports, patent filings, questionnaire responses, written correspondence, song lyrics, poems, diaries, novels, parliamentary proceedings, meeting minutes, and thousands of other forms that can be (and are) used in social science research and data mining.

Text is an excellent source of information, not just because of its scale and availability. It is also (relatively) permanent, and–most importantly–it encodes language. This one human faculty reflects (indirectly and sometimes even directly) a wide range of socio-cultural and psychological constructs: trust, power, beliefs, fears. Text analysis has consequently been used to measure socio-cultural constructs such as trust (Niculae et al., 2015) and power (Prabhakaran et al., 2012). Language encodes the age, gender, origin, and many other demographic factors of the author (Labov, 1972; Trudgill, 2000; Pennebaker, 2011). Text can, therefore, be used to measure the attitudes of society towards those target concepts over time, see (Kulkarni et al., 2015; Hamilton et al., 2016; Garg et al., 2018).

unstructured data

However, this avalanche of great data can quickly become overwhelming, and dealing with it can feel daunting. Text is often called "**unstructured data**", meaning it does not come in a spreadsheet, neatly ordered into categories. It has varying lengths, and can not readily be fed into your favorite statistical analysis tool without formatting it first. As we will see, though, "unstructured" is a bit of a misnomer. Text is by no means without any structure — it follows very regular structures, governed by syntactic rules. And if you know about these, it becomes a lot easier to make sense of text.

There are even simpler regularities in text that we can explore to get information. From simple counts to probabilities, we can do a lot to weed out the noise and gather some insights into the data. And then there are the heavy-lifters: machine learning tools that can crunch almost limitless amounts of data and extract precisely the information you want. Well, as precise as these tools can get.

This book is aimed at the working social scientist who wants to use text in their exploratory analysis. It is, therefore, first and foremost, a practical book, giving concrete examples. Each section contains Python 3.6 code using the latest available Python packages to execute the examples we will discuss. This book assumes enough

familiarity with Python to follow those examples. It does not serve as an introduction to programming.

Using off-the-shelf libraries, rather than coding the algorithms from first principles, strips away some of the complexity of the task, which makes it a lot more approachable. It also allows programming novices to use efficient code to try out a research idea, rather than spending days to make the code scale to the problem and debug it. At the same time, using packages hides some of the inner workings of the algorithms. However, I would argue that we do not need to implement the algorithms from scratch to use these tools to the full potential. It is essential, though, to know how these tools work. Knowing what is going on under the hood helps us make the right choices and to understand the possibilities and limitations. For each example, we will look at the general idea behind the algorithm, as well as the mathematical underpinnings. If you would like to dive deeper into a particular technique, or need to modify it to suit your needs, I will provide pointers to detailed tutorials and background reading.

Natural language processing (NLP), the branch of AI that applies machine learning methods to text), is becoming ubiquitous in social sciences. There is a growing number of overview articles and tutorials on text analysis and in several disciplines. These works cover the most relevant terms, models, and use cases for the respective fields. It makes sense to consult them to see what people in a specific field have been using. Some of the more recent examples include Grimmer and Stewart (2013) for political science, Evans and Aceves (2016) for sociology, Humphreys and Wang (2017) and Hartmann et al. (2018) for marketing, and Gentzkow et al. (2017) for economics.

The best starting point to dive into NLP in all its depth and glory is the textbook by Jurafsky and Martin (2014), which is constantly updated (important for such a dynamic field). The book by Manning and Schütze (1999) is still a good complement, especially on the fundamentals, despite its age and lack of some of the latest machine learning approaches. The latest textbook is Eisenstein (2019). If you are interested in specific topics, you can search the anthology of the Association for Computational Linguistics (ACL), the official organization of most NLP research. All publications in this field are peer-reviewed conference proceedings (with acceptance rates around 20–25%), and the entire catalog of all papers ever published is available for free at `https://www.aclweb.org/anthology/`. The field borrows heavily from machine learning, especially deep learning. For an excellent overview of the most relevant deep learning methods in NLP, see the book by Goldberg (2017), or the freely available primer (Goldberg, 2016) it is based on. To explore these topics in Python, the book by Marsland (2015) and Chollet (2017), respectively, are very practically oriented and beginner-friendly. For a very readable general overview of machine learning, see Murphy (2012).

This book has three parts. In the first part, we will look at some of the basic properties of text and language – the levels of linguistic analysis, grammatical and

semantic components, and how to describe them. We will also discuss what to remove and what to keep for our analyses and how to compute simple, useful statistics.

In the second part, we will look at exploration, the discovery of latent structure in the data. We will go from simple statistics to more sophisticated machine learning methods, such as topic models, word embeddings, and dimensionality reduction.

In the third part, we will cover text classification, such as sentiment analysis and other supervised machine learning techniques. We will also cover basic notions of ethics in NLP, performance measures and improvements, as well as significance testing. In the light of recent advances of deep learning techniques for NLP, those topics will receive special, separate attention.

Acknowledgements

I would like to thank all the participants of my first NLP classes at Bocconi for their detailed feedback on the first drafts of this material, and everyone who has made suggestions.

If you find a mistake, have a suggestion, or know of a missing reference, please contact me directly!

Part I

Background

1. Prerequisites

This book assumes some basic familiarity with Python. However, it does not replace an introduction to programming. If you are looking to learn Python, or to refresh the basics, the official Python website lists a number of excellent online tutorials (`https://wiki.python.org/moin/IntroductoryBooks`) and text books (`https://wiki.python.org/moin/BeginnersGuide/NonProgrammers`).

Jupyter notebooks

To develop Python programs, one of the easiest environments is that of **Jupyter notebooks**, which allow for code, text, and images to be stored in a platform-independent way. It is an excellent way to visualize and store intermediate steps. To use these notebooks, you will have to install Anaconda Python and make sure it is running. The easiest way to do so is by using the following steps:

(1) download Anaconda with Python 3.7 or higher for your operating system from `https://www.anaconda.com/download/`
(2) install Anaconda and make it is set as your system default Python
(3) open the Jupyter Notebook app either by
  (a) opening the Anaconda Navigator app and clicking on "Jupyter Notebook"
  (b) OR: opening a terminal window and typing "`jupyter notebook`"
(4) this will give you a browser window with a file structure. Make sure you know where on your computer this folder is!
(5) open a new Notebook window and copy the code from the code snippet 1 below into it
(6) execute the code in the notebook by either clicking the Play button or hitting `SHIFT+ENTER`

```
garble = 'C#LoBln{;gFZr>!a%LtVQu!(l&Ya!zt/;i iofKn: sg_,gh %gyGroUquJ
    ` jeaY!r:-e]m $taIZlbCr "e;Fa{Yd,hy>] _xog*nb{ |Ky*PoY)uTKr&K %)
   wipa@ny{g QEtMloXy %apWsr,foO\'g+RryBaD\'mT-m*}iIrnE;gy=!>t'
print(''.join([garble[c] for c in range(0, len(garble), 3)]))
```

CODE 1. Our first Jupyter notebook code.

You should see an encouraging message if all is working correctly.

spacy
nltk
gensim

In order to work with text, we will use the following libraries:

(1) **`spacy`** (including at least the English language model)
(2) **`nltk`** (including the data sets it provides)
(3) **`gensim`** (for topic models)

The `spacy` library should come pre-installed in Anaconda, but you will have to download the models for various languages. You can check this by executing the following code in a notebook cell:

```
import spacy
spacy.load('en')
```

CODE 2. Loading the English language model of `spacy`.

If you get an error, install at least the English model (there are additional languages available). See `https://spacy.io/usage/models#section-install` for details on how to do this.

You can install the gensim and nltk library through Anaconda–which we will need for embeddings and topic models–either through the Anaconda Navigator *or* the terminal by executing the following lines:

```
conda install nltk
conda install gensim
```

In order to install the NLTK data sets, you can simply execute the instructions in the following code cell in a notebook:

```
import nltk
nltk.download('all')
```

CODE 3. Downloading the data sets of `nltk`.

2. What's in a Word

Say we have collected a large number of companies' self-descriptions from their websites. We want to start finding out whether there are any systematic differences, what the big themes are, and how the companies typically act. The question is: *How do we do all that*? Does it matter that firms *buy*, *bought*, and *are buying* – or do we only want to know that there is any buying event, no matter when? Do we care about prepositions (e.g., *in*, *off*, *over*, *about*, etc.), or are they just distracting us from what we really want to know? How do we tell the program that "Facebook acquires Whatsapp" and "Whatsapp acquired by Facebook" mean the same thing, but "Whatsapp acquires Facebook" does not (even though it uses the same words as the first example)?

Before we dive into the applications, let's take a look at the subject we are working with: language. In this section, we will look at the terminology to describe some of its basic elements (morphology, syntax, and semantics), and their equivalents in text (characters, words, sentences). To choose the right method for any text-related research question we have, it makes sense to think about what language is and what it is not, how it is structured, and how it "works". This section does not replace an introduction to linguistics, but it gives us a solid starting point for our purposes. If you are interested in reading more on this, there are many excellent introductory textbooks to linguistics and its diverse subfields. One of the most entertaining ones is Fromkin et al. (2018). For a more focused overview of English and its history, see Crystal (2003).

Language often encodes information redundantly, i.e., we say the same thing in several ways: through the meaning of the words, their position, the context, and many other cues. Words themselves consist of different components, which are the focus of different linguistic disciplines: their meaning (**semantics**), their function in a sentence (**syntax**), the prefixes and endings (**morphology**). Not all words have all of this information. And when we work with textual data, we might not be interested in all of this information. In fact, it can be beneficial to remove some of the information we do not need.

semantics
syntax
morphology

When we work with text, the unit we are interested in depends strongly on the problem we investigate. Traditionally, this was a report or article. However, when we work with social media, it can also refer to a user's entire posting history, to a single message, or even to an individual sentence. Throughout this Element, we will refer to all these units of text as **documents**. It should be clear from the context what size a document has. Crucially, one document always represents one observation in our data. To refer to the entire collection of documents/observations, we use the word **corpus** (plural *corpora*).

documents
corpus

The set of all the unique terms in our data is called the **vocabulary** (V). Each element in this set is called a **type**. Each *occurrence* of a type in the data is called a **token**. So the sentence "*a good sentence is a sentence that has good words*" has 10 tokens, but

vocabulary
type
token

only 7 types (namely "*a*", "*good*", "*sentence*", "*is*", "*that*", "*has*", and "*words*"). Note that types can also include punctuation marks and multi-word expressions (see more in Section 6).

2.1. Word Descriptors.

2.1.1. *Tokens and Splitting.* Imagine we are looking at reports and want to filter out short sentences because they don't contain anything of interest. The easiest way to do so is by defining a cutoff for the number of words. However, it can be surprisingly tricky to define what a **word** is. How many words are there in "She went to Berlin," and in "She went to San Luis Obispo"? The most general definition used in many languages is any string of characters delimited by white space. However, note that Chinese, for example, does not use white space between words. Unfortunately, not all words are surrounded by white space. Words at the beginning of a line have no white space beforehand. Words at the end of a phrase or sentence might have a punctuation symbol (commas, full stops, exclamation or question marks, etc.) directly attached to them. Quotation marks and brackets complicate things even more.

word

Of course, we can separate these symbols by introducing extra white space. This process is called **tokenization** (because we make each word and punctuation mark a separate token). There is a different process, called **sentence splitting**, which separates a document into sentences and works similarly. The problem there is to decide when a dot is part of a word (as in titles like *Mr.* or abbreviations like *abbr.*), or a full stop at the end of a sentence. Both tokenization and sentence splitting are prediction tasks, for which there exist reliable machine learning models. We will not discuss the inner workings of these tools in detail here, but instead rely on the available Python implementations in the `spacy` library.

tokenization

sentence splitting

We need to load the library, create an instance for the language we work with (here, English), and can then use it on any input string.

```
import spacy
nlp = spacy.load('en')

documents = "I've been 2 times to New York in 2011, but did not have the constitution for it. It DIDN'T appeal to me. I preferred Los Angeles."
tokens = [[token.text for token in sentence] for sentence in nlp(documents).sents]
```

CODE 4. Applying sentence splitting and tokenization to a set of document.

Executing the example in Code 4 gives us the following results for `tokens`:

```
[['I', "'ve", 'been', '2', 'times', 'to', 'New', 'York', 'in', '2011', ',', 'but', 'did', 'not', 'have', 'the', 'constitution', 'for', 'it', '.'],
 ['It', "DIDN'T", 'appeal', 'to', 'me', '.'],
```

```
3 ['I', 'preferred', 'Los', 'Angeles', '.']]
```

Notice that the capitalized word *DIDN'T* was not tokenized properly, but the first, lowercased, version was. This difference suggests a common and simple solution, namely converting everything to lower case as a preprocessing step.

2.1.2. *Lemmatization.* Suppose we have a large corpus of articles from the business section of various newspapers. We are interested in how often and which companies acquire each other. We have a list of words, but words come in different forms, depending on the tense and aspect, so we might have to look for "acquire", "acquires", "acquired", and "acquiring" (since we are dealing with newspaper articles, they might be referring to recent events, to future plans, and they might quote people). We could try to formalize this pattern by just looking for the endings *–e*, *–es*, *–ed*, and *–ing*, but that only works for **regular verbs** ending in *–re*. If we are interested in **irregular verbs**, we might see forms like "go", "goes", "went", "gone", or "going". And it does not stop there: the firms we are looking for might acquire just one "subsidi*ary*", or several "subsidi*aries*", one "comp*any*" or several "comp*anies*". We need a more principled way to deal with this variation.

regular verbs

irregular verbs

When we look up a word in a dictionary, we usually just look for the **base form** (in the previous example, that would be the infinitive, "go"). This dictionary base form is called the **lemma**. All the other forms don't change the core meaning of this lemma, but add further information (such as temporal and other aspects). Many of these inflections are required by the syntax, i.e., the context and word order that make a sentence grammatical. When we work with text and are more interested in the meaning, rather than the morphology or syntax, it can be useful to replace each word with its lemma. This step reduces the amount of variation in the data, and makes it easier to collect meaningful statistics. Rather than getting a single count for each of "go", "goes", "went", "gone", and "going", we would simply record having seen the form "go" five times in our data. This reduction does lose some of the temporal and other syntactic information, but it might be irrelevant for our purposes. Many words can be lemmatized by undoing certain patterns, based on the type of word (e.g., remove "–ed" from the end of a verb like "walked", but remove "–ier" from the end of an adjective like "happier"). For the exceptions (for example, "to go"), we have to have a lookup table.

base form

lemma

Luckily, **lemmatization** is already built into `spacy`, so we can make use of it. Applying lemmatization to our example sentences from above:

lemmatization

```
1 lemmas = [[token.lemma_ for token in sentence] for sentence in nlp(
    documents).sents]
```

CODE 5. Applying lemmatization to a set of documents.

we get

```
[['-PRON-', 'have', 'be', '2', 'time', 'to', 'new', 'york', 'in', '
   2011', ',', 'but', 'do', 'not', 'have', 'the', 'constitution', '
   for', '-PRON-', '.'],
 ['-PRON-', "didn't", 'appeal', 'to', '-PRON-', '.'],
 ['-PRON-', 'prefer', 'los', 'angeles', '.']]
```

Note that as part of the process, all words are converted to lower case (if we have not done so already ourselves), while pronouns are all conflated as a special token -PRON- (see also next section, 2.2). Again, the word *DIDN'T* was not lemmatized correctly, but simply changed to lowercase.

2.1.3. *Stemming*. Maybe we are searching legal texts for political decisions, and we are generally interested in anything that has to do with the constitution. So the words "constitution", "constitutions", "constitutional", "constitutionality", and "constitutionalism" are all of interest to us, despite having different parts of speech. Lemmatization will not help us here, so we need another way to group them across word classes.

An even more radical way to reduce variation is **stemming**. Rather than reducing a word to the lemma, we strip away everything but the irreducible morphological core (the **stem**). E.g., for a word like "anticonstitutionalism", which can be analyzed as "anti+constitut+ion+al+ism", we remove everything but "constitut". The most famous and commonly used stemming tool is based on the algorithm developed by Porter (1980). For each language, it defines a number of **suffix**es (i.e., word endings), and the order in which they should be removed or replaced. By repeatedly applying these actions, we reduce all words to their stem. In our example, all words derive from the stem "constitut–", by attaching different endings.

stemming

stem

suffix

Again, a version of the Porter stemmer is already available in Python, in the `nltk` library (Loper and Bird, 2002), but we have to specify the language:

```
from nltk import SnowballStemmer
stemmer = SnowballStemmer('english')
stems = [[stemmer.stem(token) for token in sentence] for sentence in
   tokens]
```

CODE 6. Applying stemming to a set of documents.

This gives us

```
[['i', 've', 'been', '2', 'time', 'to', 'new', 'york', 'in', '2011',
   ',', 'but', 'did', 'not', 'have', 'the', 'constitut', 'for', 'it',
    '.'],
 ['it', "didn't", 'appeal', 'to', 'me', '.'],
 ['i', 'prefer', 'los', 'angel', '.']]
```

While this is extremely effective in reducing variation in the data, it can make the results harder to interpret. One way to address this problem is to keep track of all the original

words that share a stem, and how many times each of them occurred. We can then replace the stem with the most common derived form (in our example, this would most likely be "constitution"). Note that this can conflate word classes, say nouns and verbs, so we need to decide whether to use this depending on our goals.

2.1.4. *n-Grams.* Looking at words individually can be a good start. Often, though, we want to look also at the immediate context. In our example, we have two concepts that span two words, "New York" and "Los Angeles", and we do not capture them by looking at each word in turn.

n-gram
unigrams
bigrams
trigrams

Instead of looking at each word in turn, we can use a sliding window of n words to examine the text. This window is called an **n-gram**, where n can have any size. Individual words are also called **unigrams**, whereas combinations of two or three words are called **bigrams** and **trigrams**, respectively. For larger n, we simply write the number, e.g., "4-grams".

We can extract n-grams with a function from `nltk`:

```
from nltk import ngrams
bigrams = [gram for gram in ngrams(tokens[0], 2)]
```

CODE 7. Extracting bigrams from a sentence.

This gives us

```
[('I', "'ve"),
 ("'ve", 'been'),
 ('been', '2'),
 ('2', 'times'),
 ('times', 'to'),
 ('to', 'New'),
 ('New', 'York'),
 ('York', 'in'),
 ('in', '2011'),
 ('2011', ','),
 (',', 'but'),
 ('but', 'did'),
 ('did', 'not'),
 ('not', 'have'),
 ('have', 'the'),
 ('the', 'constitution'),
 ('constitution', 'for'),
 ('for', 'it'),
 ('it', '.')]
```

We will see later how we can join frequent bigram expressions that are part of the same "word" (see section 6).

2.2. Parts of Speech. Let's say we have collected a large corpus of restaurant menus, including both fine dining and fast-food joints. We want to know how each

of them describes the food. Is it simply "grilled", or is it "juicy", "fresh", or even "artisanal"? All of these food descriptors are **adjectives**, and by extracting them and correlating them with the type of restaurant who use them, we can learn something about the restaurants' self-representation – and about the correlation with price.[1]

adjectives

At a very high level, words denote things, actions, and qualities in the world. These categories correspond to **parts of speech**, e.g., nouns, verbs, and adjectives (most languages have more categories than just these three). They are jointly called **content words** or **open-class words** because we can add new words to each of these categories (for example, new nouns, like "Tweet", or verbs like "twerking"). There are other word classes, like determiners, prepositions, etc. (see below). They don't "mean" anything (i.e., they are not referring to a concept) but help to structure a sentence and to make it grammatical. They are therefore referred to jointly as **function words** or **closed-class words** (it is very unlikely that anybody comes up with a new preposition any time soon). Partially because function words are so short and ubiquitous, they are often overlooked. While we have a rough idea of how many times we have seen the noun "class" in the last few sentences, it is almost impossible to consciously notice how often we have seen, say, "in". [2]

parts of speech

content words
open-class words

function words
closed-class words

Languages differ in the way they structure sentences. Consequentially, there was little agreement about the precise number of these grammatical categories, beyond the big three of content words (and even those are not always sure). The need for NLP tools to work across languages recently spawned efforts to come up with a small set of categories that applies to a wide range of languages (Petrov et al., 2011). It is called the Universal Part-of-Speech tag set (see `https://universaldependencies.org/u/pos/`). In this book, we will use this set of 15 parts of speech.

Open-class words:

- ADJ: adjectives. They modify nouns to specify their properties. Examples: *awesome*, *red*, *boring*
- ADV: adverbs. They modify verbs, but also serve as question markers. Examples: *quietly*, *where*, *never*
- INTJ: interjections. Exclamations of some sort. Examples: *ouch*, *shhh*, *oi*
- NOUN: nouns. Entities in the world. Examples: *book*, *war*, *shark*

[1] This example is based on the book "The Language of Food: A Linguist Reads the Menu", by Dan Jurafsky. He did exactly this analysis, and found that the kinds of adjectives used are a good indicator of the price the restaurant charges (though maybe not necessarily of the quality).

[2] Even though we do not notice function words, we tend to have individual patterns for using them. This property makes them particularly interesting for psychological analysis and forensic linguistics. The book "The Secret Life of Pronouns: What Our Words Say About Us" by James Pennebaker examines this quality of function words. They can therefore be used to identify kidnappers, predict depression, or assess the stability of a marriage.

- PROPN: proper nouns. Names of entities, a subclass of nouns. Examples: *Rosa*, *Twitter*, *CNN*
- VERB: full verbs. Events in the world. Examples: *codes*, *submitted*, *succeed*

Closed-class words:

- ADP: adpositions. Prepositions or postpositions, markers of time, place, beneficiary, etc. Examples: *over*, *before*, *(get) down*
- AUX: auxiliary and modal verbs. Used to change time or modality. Examples: *have (been)*, *could (do)*, *will (change)*
- CCONJ: coordinating conjunctions. Link together parts of sentences with equal importance. Examples: *and*, *or*, *but*
- DET: determiners. Articles and quantifiers. Examples: *a*, *they*, *which*
- NUM: numbers. Exactly what you would think it is...
- PART: particles. Possessives and grammatical markers. Examples: *'s*
- PRON: pronouns. Substitutions for nouns. Examples: *you*, *her*, *his*, *myself*
- SCONJ: subordinating conjunctions. Link together parts of sentences with one part being more important. Examples: *since*, *if*, *that*

Other

- PUNCT: punctuation marks. Examples: *!*, *?*, –
- SYM: symbols. Word-like entities, often special characters, including emojis. Examples: *%*, *$*, *:)*
- X: other. Anything that does not fit into any of the above. Examples: *pfffrt*

POS tagging
parsing

Automatically determining the parts of speech and syntax of a sentence are known as **POS tagging** and **parsing**, two of the earliest and most successful NLP applications. We can again use the POS-tagger in `spacy`:

```
pos = [[token.pos_ for token in sentence] for sentence in nlp(
    documents).sents]
```

CODE 8. POS-tagging a set of documents.

This gives us

```
[['PRON', 'VERB', 'VERB', 'NUM', 'NOUN', 'ADP', 'PROPN', 'PROPN', '
    ADP', 'NUM', 'PUNCT', 'CCONJ', 'VERB', 'ADV', 'VERB', 'DET', 'NOUN
    ', 'ADP', 'PRON', 'PUNCT'],
 ['PRON', 'PUNCT', 'VERB', 'ADP', 'PRON', 'PUNCT'],
 ['PRON', 'VERB', 'PROPN', 'PROPN', 'PUNCT']]
```

Because POS tagging was one of the first successful NLP applications, a lot of research has gone into it. By now, POS taggers are more accurate, more consistent, and definitely much faster than even the best-trained linguists.

2.3. Stopwords. If we want to assess the general topics in product reviews, we usually do not care whether a review refers to "*the* price" or "*a* price", and can remove the determiner. Similarly, if we are looking at the political position of parties in their

manifestos, we know who wrote for each manifesto. So we do not need to keep their names when they mention themselves in the document. In both cases, we have a set of words that occur often, but do not contribute much to our task, so it can be beneficial to remove them. The set of these ignorable words is called **stopwords**. **stopwords**

As we have seen above, many words in a text belong to the set of function words. In fact, the most-frequent words in any language are predominantly function words. The ten most common words in English (according to the Oxford English Corpus), with their parts of speech are: *the* (DET), *be* (VERB/AUX), *to* (ADP/PART), *of* (ADP), *and* (CCONJ), *a* (DET), *in* (ADP), *that* (CCONJ), *have* (VERB/AUX), *I* (PRON). While these are all very useful words when building a sentence, they do not really mean much on their own, i.e., without context. In fact, for most applications (e.g., topic models, see Boyd-Graber et al. (2014)), it is better to ignore them altogether.[3]

We can either exclude stopwords based on their part of speech (see above), or by using a list. The former is more general, but it risks throwing out some unintended candidates. (If we exclude prepositions, we risk losing the noun "round" if it gets incorrectly labeled.) The latter is more task-specific (e.g., we can use a list of political party names), but it risks leaving out words we were not aware of when compiling the list. Often, it can therefore be beneficial to use a combination of both. In `spacy`, we can use the `is_stop` property of a token, which checks it against a list of common stopwords. The list of English stopwords is in Appendix 2.

For our running example, if we exclude common stopwords, and filter out non-content words (all parts of speech except NOUN, VERB, PROPN, ADJ, and ADV):

```
content = [[token.text for token in sentence
        if token.pos_ in {'NOUN', 'VERB', 'PROPN', 'ADJ', 'ADV'}
        and not token.is_stop]
        for sentence in nlp(documents).sents]
```

CODE 9. Selecting content words based on POS.

This gives us

```
[["'ve", 'times', 'New', 'York', 'constitution'],
 ['appeal'],
 ['preferred', 'Los', 'Angeles']]
```

Another way to reduce variation is to replace any numbers with a special token, rather than to remove them. There are infinitely many possible numbers, and in many contexts, we do not care about the exact amount they denote (unless, of course, we are interested in prices). We will see an example of how to do this most efficiently in section 4, when we learn about regular expressions.

[3] See the exception for profiling above, though.

2.4. Named Entities. Say we are interested in the changing patterns of tourism and want to find the most popular destinations over time in a corpus of travel blogs. We would like a way to identify the names of countries, cities, and landmarks. Using POS tagging, we can easily identify all proper names, but that still does not tell us with what kind of entity we are dealing. These semantic categories of words are referred to as **named entities**.

named entities

Proper names refer to entities in the real world, such as companies, places, persons, or something else: e.g., *Apple*, *Italy*, *George Oscar Bluth*, *LAX* or *Mercedes*. While implementing a **named entity recognizer** (NER) is outside the scope of this work, it generally works by using a list of known entities that are unlikely to change (for example, country names or first names). However, names are probably the most open-ended and innovative class of words, and there are plenty of entities that are not listed anywhere. Luckily, we can often determine what kind of entity we have by observing the context in which they occur. E.g., company names might be followed by "Inc.", "LLC", etc. Syntax can give us more cues—if an entity *says* something, *eats*, *sleeps*, or is associated with other verbs that denote human activities, we can label it as a person. (Of course, this gets trickier when we have metaphorical language, e.g., "parliament says...").

named entity recognizer

Luckily, `spacy` provides a named entity recognizer that can help us identify a wide range of types: PERSON, NORP (Nationality OR Religious or Political group), FAC (facility), ORG (organization), GPE (GeoPolitical Entity), LOC (locations, such as seas or mountains), PRODUCT, EVENT (in sports, politics, history, etc.), WORK_OF_ART, LAW, LANGUAGE, DATE, TIME, PERCENT, MONEY, QUANTITY, ORDINAL (ordinal numbers such as "third", etc.), and CARDINAL (regular numbers).

For our running example, we can use the following code to extract the words and their types:

```
entities = [[(entity.text, entity.label_)
             for entity in nlp(sentence.text).ents]
             for sentence in nlp(documents).sents]
```

CODE 10. Named Entity Recognition.

This gives us

```
[[('2', 'CARDINAL'), ('New York', 'GPE'), ('2011', 'DATE')],
 [],
 [('Los Angeles', 'GPE')]]
```

2.5. Syntax.

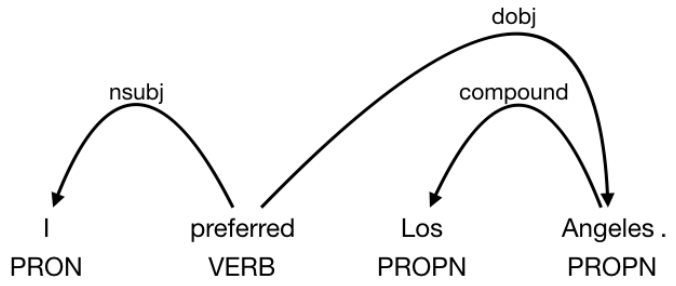


FIGURE 1. Example of a dependency parse.

Let's say we are interested in firm acquisitions. Imagine we want to filter out all the business transactions (who bought whom) from a large corpus of newswire text. We define a list of verbs that express acquisitions and want to find all the nouns associated with these verbs. I.e., who is doing the acquiring (the subjects), or who is being acquired (the objects). We want to end up with NOUN-VERB-NOUN triples that give us the names of the firms and other entities involved in the acquisitions, as well as the verb.

We have already seen that words can have different grammatical forms, i.e., parts of speech (nouns, verbs, etc.) However, a sentence can have many words with the same POS, and what a specific word means in a sentence depends in part on its **grammatical function**. Sentence structure, or **syntax** is an essential field of study in linguistics, and explains how words in a sentence hang together.

grammatical function
syntax

One of the most important cues to a word's function in English is its position in the sentence. In English, grammatical function is mainly determined by word order, while other languages use markers on the words. Changing the word order changes the grammatical function of the word. Consider the difference between "Swimmer eats fish," and "Fish eats swimmer". The **subject** (the entity doing the eating) and the **object** (the entity being eaten) are now switched. We have already seen verbs, subjects, and objects as examples of syntactic functions, but there are many other possible **dependency relations** that can hold between words. The idea in dependency grammar is that the sentence "hangs" off the main verb like a mobile. The links between words describe how the words are connected. You can see an example in Figure 1.

subject
object
dependency relations

Naturally, these connections can vary quite a bit between languages, but similar to the Universal POS tags, there have been efforts to unify and limit the set of possibilities (McDonald et al., 2013; Nivre et al., 2015, 2016, inter alia). Here is a list of the ones used in the Universal Dependencies project (again, see `https://universaldependencies.org`):

- `acl`, clausal modifier of a noun (adjectival clause)
- `advcl`, adverbial clause modifier
- `advmod`, adverbial modifier
- `amod`, adjectival modifier
- `appos`, appositional modifier
- `aux`, auxiliary verb
- `case`, case marker
- `cc`, coordinating conjunction
- `ccomp`, clausal complement
- `clf`, classifier
- `compound`, compound
- `conj`, conjunction
- `cop`, copula
- `csubj`, clausal subject
- `dep`, unspecified dependency
- `det`, determiner
- `discourse`, discourse element
- `dislocated`, dislocated elements
- `dobj`, direct object
- `expl`, expletive
- `fixed`, fixed multiword expression
- `flat`, flat multiword expression
- `goeswith`, goes with
- `iobj`, indirect object
- `list`, list
- `mark`, marker
- `nmod`, nominal modifier
- `nsubj`, nominal subject
- `nummod`, numeric modifier
- `obj`, object
- `obl`, oblique nominal
- `orphan`, orphan
- `parataxis`, parataxis
- `pobj`, prepositional object
- `punct`, punctuation
- `reparandum`, overridden disfluency
- `root`, the root of the sentence, usually a verb
- `vocative`, vocative
- `xcomp`, open clausal complement

Many of these might never or rarely occur in the language we study, and for many applications, we might only want to look at a small handful of relations, e.g. the core

arguments, `nsubj`, `obj`, `iobj`, `pobj`, or `root`. We can see an example of a dependency parse in Figure 1. Why is this relevant? Because knowing where a word stands in the sentence and what other words are related to it (and in which function), helps us make sense of a sentence.

In Python, we can use the parser from the `spacy` library to get

```
[[(c.text, c.head.text, c.dep_) for c in nlp(sentence.text)]
 for sentence in nlp(documents).sents]
```

CODE 11. Parsing.

For the first sentence, this gives us

```
[('I', 'been', 'nsubj'),
  ("'ve", 'been', 'aux'),
  ('been', 'been', 'ROOT'),
  ('2', 'been', 'npadvmod'),
  ('times', '2', 'quantmod'),
  ('to', '2', 'prep'),
  ('New', 'York', 'compound'),
  ('York', 'to', 'pobj'),
  ('in', 'been', 'prep'),
  ('2011', 'in', 'pobj'),
  (',', 'been', 'punct'),
  ('but', 'been', 'cc'),
  ('did', 'have', 'aux'),
  ('not', 'have', 'neg'),
  ('have', 'been', 'conj'),
  ('the', 'constitution', 'det'),
  ('constitution', 'have', 'dobj'),
  ('for', 'have', 'prep'),
  ('it', 'for', 'pobj'),
  ('.', 'been', 'punct')]
```

We can see how the verb "been" is the root node that everything else hangs off of. For our example with the firm acquisitions, we would want to extract anything labeled with `nsubj`, `dobj`, or `pobj`.

2.6. Caveats – What if it's not English? Say you have a whole load of Italian data that you want to work with, doing some of the things we have done in the previous sections. What are your options?

`spacy` comes with support for a number of other languages, including German (`de`), Spanish (`es`), French (`fr`), Italian (`it`), Dutch (`nl`), and Portuguese (`pt`). All you have to do is load the correct library:

```
import spacy
nlp = spacy.load('it')
```

However, you need to download these language models separately and make sure they are in the right path. They mostly offer all the same functionalities (lemmatization, tagging, parsing). However, their performance can vary. Because NLP has been so focused on English, there is often much less training data for other languages. Consequently, the statistical models are often much less precise.

NLP was developed predominantly in the English-speaking world, by people working on English texts. However, English is not like many other languages. It is a pidgin language between Celtic, old Germanic languages, Franco-Latin, and Norse, and at some point it lost most of its inflections (the word endings that we learn in declension tables). However, word endings allow you to move the word order around: you still know from the endings what the subject and what the object is, no matter where they occur. To still distinguish subject from object, English had to adopt a fixed word order. Because of this historical development, n-gram methods work well for English: we see the same order many times can get good statistics over word combinations. We might not see the same n-gram more than once in languages that are highly inflected or have variable word order — no matter how large our corpus is. Finnish, for example, has 15 cases, i.e., many more possible word endings, and German orders subordinate clauses differently from main clauses. For those languages, lemmatization becomes even more important, and we need to rely on syntactic n-grams rather than sequences.

Alternatively, `nltk` offers support to stem text in Danish, Dutch, English, Finnish, French, German, Hungarian, Italian, Norwegian, Portuguese, Romanian, Russian, Spanish, and Swedish. This method is, of course, much coarser than lemmatization, but it is an efficient way to reduce the noise in the data.

There are programs out there to tag and parse more languages, most notably the TreeTagger algorithm, which can be trained on data from the Universal POS project. However, those programs involve some manual installation and command-line running that go beyond the scope of this book.

If you are working with other languages than English, you can, in many cases, still lemmatize the data. However, you might have fewer or no options for POS tagging, parsing, and NER. For the applications in the following sections, though, adequate preprocessing is often more important than being able to perform all types of linguistic analyses. However, if your analysis relies on these tools, take a look at the NLP literature on low-resource language processing, a thankfully growing subfield in NLP.

3. Representing Text

Say you have collected a large corpus of texts on political agendas and would like to work with it. As a human, you can just print them out, take a pen, and start to take notes. But how do you let a computer do the same?

To work with texts and gain insights, we need to represent the documents in a way for the computer to process it. We cannot simply take a Microsoft Word document and work with that. (Word documents are great for writing but terrible to analyze, since they store a lot of formatting information along with the content.) Instead, we need to get it into a form that computers can understand and manipulate. Typically, this is done via representing documents as **feature vectors**. Vectors are essentially lists of numbers, but they have several useful properties that make them easy to work with. The domain of working with vectors and matrices is **linear algebra**. We will use many of the concepts from linear algebra, so it can be good to refresh your knowledge. A good place to start is the YouTube channel 3Blue1Brown, which has a dedicated series on the topic (`https://www.youtube.com/playlist?list=PLZHQObOWTQDPD3MizzM2xVFitgF8hE_ab`). Lang (2012) is a great textbook. We will only cover the basics here. Vectors can either represent a collection of **discrete features**, or a collection of **continuous dimensions**. Discrete features characterize a word or a document (in which case they are called **discrete representations**). The dimensions of **distributed representations** represent the word or document holistically, i.e., their position relative to all other words or documents in our corpus.

feature vectors

linear algebra

discrete features

continuous dimensions

discrete representations

distributed representations

In discrete feature vectors, each dimension of the vector represents a feature with a specific meaning (e.g., the fifth dimension means whether the word contains a vowel or not). The values in the vector can either be binary indicator variables (0 or 1) or some form of counts. Discrete representations are often also referred to as **sparse** vectors, because not every document will have all the features. For the most part, these vectors are empty (e.g., a document might not contain the word `tsk!`, so the corresponding position in the vector will be empty). We will see discrete and sparse vectors when we look at bags of words, counts, and probabilities in section 3.2.

sparse

In distributed or continuous representations, vectors as a whole represent points in a high-dimensional space, i.e., by a vector of fixed dimensionality with continuous-valued numbers. The individual dimensions of the vectors do not mean anything anymore, i.e., they can not be interpreted. Instead, they describe the coordinates of a document relative to all other documents in the vector space. These vector representations are also called **dense**, because we do not have any empty positions in the vectors, but will always have some value in each position. When the vectors represent words or documents, these high-dimensional, dense vector representations are called **embeddings** (because they *embed* words or documents in the high-dimensional space). We will see distributed representations when we look at word and document embeddings in Section 3.3.

dense

embeddings

3.1. Enter the Matrix. After you choose how to represent each document in your corpus, you end up with a bunch of vectors. Put those vectors on top of each other, and you have a matrix. Let's talk about some of the terminology.

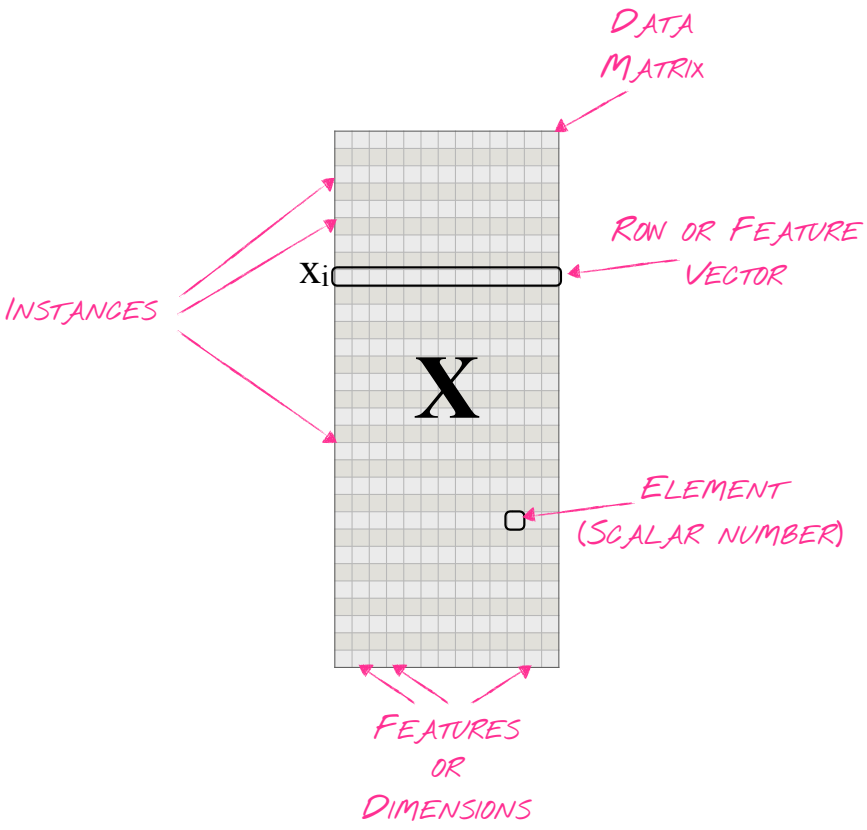


FIGURE 2. Elements of the data matrix.

matrices and vectors

Irrespective of the type of representation we choose (sparse or dense), it makes sense to think of the data in terms of **matrices and vectors**. A vector is an ordered collection of D dimensions with numbers in them. Each dimension is a feature in discrete vectors. A matrix is a collection of N vectors, where each row represents one document, and the columns represent the features mentioned above, or the dimensions in the embedding space. This matrix is also sometimes referred to as **term-document matrix** (Manning and Schütze, 1999), reflecting the two elements involved in its construction. When we refer to the **data matrix** in the following, we mean a matrix with one row for each document, and as many columns as we have features or dimensions in our representation. If we want to look at one particular feature or dimension, we can thus extract the corresponding **column vector**. And when we refer to a document, unless specified otherwise, we refer to a **row vector** in the data matrix. See Figure 2 for an illustration of these elements.

term-document matrix

data matrix

column vector

row vector

We generally refer to **inputs** to our algorithms with X, to **outputs** with Y. We will use these uppercase symbols to refer to vectors. If the input is a single **scalar** number, we use lowercase x or y. If we refer to a specific element of a vector, we will use an index to refer to the position in the vector, e.g., x_i, where i is an integer $\in D$ denoting the specific dimension. For matrices, we will use bold-faced uppercase $\mathbf{X}$, and denote individual elements with their row and column index: $x_{i,j}$. Generally, we will use $i \in 1..N$ to iterate over the documents, and $j \in 1..D$ to iterate over the dimensions.

inputs
outputs
scalar

Thinking of the data this way allows us to use linear algebra methods. We can now compute the linear combination of the data with some coefficients. Or we can decompose the data, reduce their dimensionality, transform it, etc. Linear algebra also allows us to combine the data with other kinds of information. For example, networks can be represented as matrices, where each row and column represents a node, and each cell tells us whether two nodes are connected (and if so, how strongly). Luckily, Python allows us to derive matrix representations from text, and to use them in various algorithms.

3.2. Discrete Representations. Suppose we have a corpus of documents, and have preprocessed it so that it contains a vocabulary of 1000 words. We want to represent each document as the number of times we have seen each of these 1000 words in a document. How do we collect the counts to make each document a 1000-dimensional vector?

The simplest way of quantifying the importance of a particular word is to count its occurrences in a document. Say we want to track the importance of a particular issue (say, "energy") for a company in their quarterly reports. Getting the **term frequency** of the (lemmatized) word can give us a rough impression of its importance over time. There is evidence that our brain keeps track of how often you have seen or heard a word as well. Well, kind of. You can immediately tell whether you have seen the word "dog" more frequently than the word "platypus", but maybe not *how* often.

term frequency

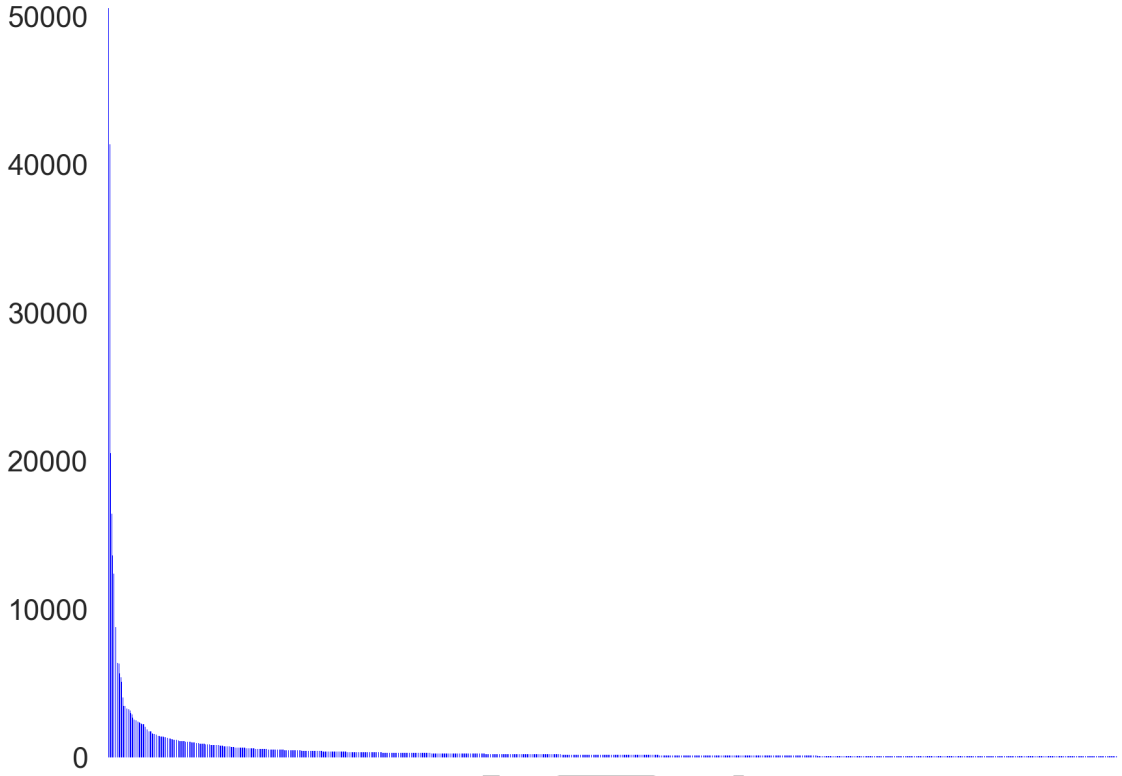


FIGURE 3. Frequency distribution of the top 1000 words in a random sample of tweets, following Zipf's law.

There are other frequency effects in language: not all words occur nearly at the same frequency. One thing to notice for word counts is their overall distribution. The most frequent word in any language is typically several times more frequent than the second most frequent word. That word, in turn, is magnitudes more frequent than the third most frequent word, etc., until we reach the "tail" of the distribution. That tail is a long list of words that occur only once (see Figure 3). This phenomenon (which is a power-law distribution) was first observed by Zipf (1935), and is therefore often referred to as **Zipf's law**. It also holds for city sizes, income, and many other socio-cultural phenomena. It is, therefore, essential to keep in mind that methods that assume a normal distribution do not work well for many, if not most, linguistic problems. Zipf's law also means that a small minority of word types account for the majority of word tokens in a corpus.[4]

Zipf's law

[4]More than half of all word tokens in a newspaper belong to a tiny group of word types. Unfortunately, those are usually not the informative word types. Which is why learning vocabulary is such an essential part of any language.

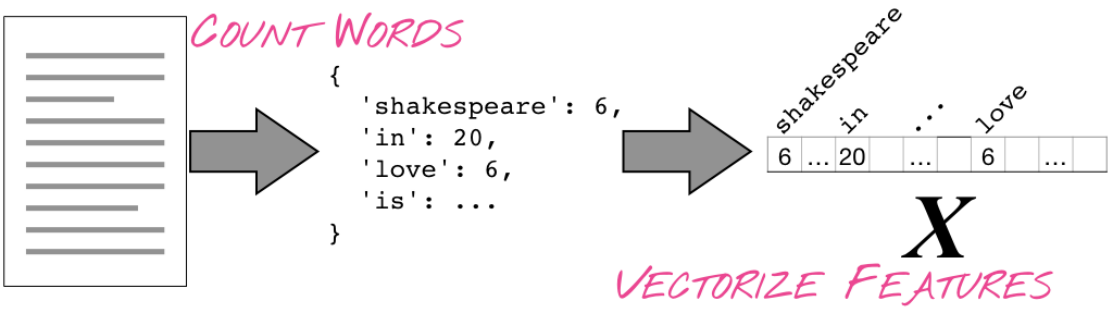


FIGURE 4. Schematic of a bag-of-words representation

One of the most important choices we have to make when working with texts is how to represent our input X. The most straightforward way to represent a document is simply to create a count vector. Each dimension represents one word in our vocabulary, so D has the size of all words in our vocabulary V, i.e., $D = |V|$. We then simply count for each document how often each word occurs in it. This representation (a **count vector**) does not pay any attention to where in the document a word occurs, what its grammatical role is, or other structural information. It simply treats a document as a bunch of word counts, and is therefore called a **bag of words** (or BOW, see Figure 4). The result of this application to our corpus is a **count matrix**.

count vector

bag of words

count matrix

In Python, we can collect counts with a function in `sklearn`:

```
from sklearn.feature_extraction.text import CountVectorizer

vectorizer = CountVectorizer(analyzer='word')

X = vectorizer.fit_transform(documents)
```

CODE 12. Extracting n-gram count representations.

3.2.1. *n-gram Features.* In addition to individual words, we would also like to account for n-grams of lengths 2 and 3, to capture both word combinations and some grammatical sequence effects in our representations. We want to be able to see the influence of "social media" as well as "social" and "media." (We will see later, in section 6, how to preprocess such terms to make them a single token.)

N-grams are sequences of tokens (either words or characters), and the easiest way to represent a text is by keeping track of the frequencies of all the n-grams in the data. Luckily, `CountVectorizer` allows us to extract all n-gram counts within a specified range.

```
from sklearn.feature_extraction.text import CountVectorizer

vectorizer = CountVectorizer(analyzer='word',
                             ngram_range=(1, 3),
```

```
                            min_df=10,
                            max_df=0.75,
                            stop_words='english')

X = vectorizer.fit_transform(documents)
```

CODE 13. Extracting n-gram count representations.

In the code in snippet 13, we create a vectorizer that collects all word unigrams, bigrams, and trigrams. To be included, these n-grams need to occur in at least ten documents, but not in more than 75% of all documents. We exclude any words that are contained in a list of stopwords.

The exact n-gram range, minimum and maximum frequency, and target entity (`'word'` or `'char'`, for character) depend on the task, but sensible starting settings are

```
word_vectorizer = CountVectorizer(analyzer='word',
                                  ngram_range=(1, 2),
                                  min_df=10,
                                  max_df=0.75,
                                  stop_words='english')

char_vectorizer = CountVectorizer(analyzer='char',
                                  ngram_range=(2, 6),
                                  min_df=10,
                                  max_df=0.75)
```

Using character n-grams can be useful when we look at individual words as documents, or if we are dealing with multiple languages in the same corpus. N-grams also help account for neologisms (new word creations), and misspellings. From a linguistic perspective, this approach may seem counterintuitive. However, from an algorithmic perspective, character n-grams have many advantages. Many character combinations occur across languages (e.g., "en"), so we can make them comparable even if we cannot use the appropriate lemmatization. N-grams capture overlap and similarities between words better than a whole-word approach. This property can help us with both spelling mistakes and new variations of words because they account for the overlap with the original word. Compare the words *definitely* and its frequent misspelling *definately*. They do have significant overlap, both in terms of form and meaning. Trigrams, for example, capture this fact: seven of the nine unique trigrams we can extract from the two words are shared. Their character count-vectors will be similar – despite the spelling variation. In a word-based approach, these two tokens would not be similar. We usually only need to store a limited number of all possible character combinations, because many do not occur in our data (e.g., "qxq"). N-grams are, therefore, also a much more efficient representation computationally: there are much fewer character combinations than there are word combinations.

3.2.2. *Syntactic Features.* One problem with n-grams is that we only capture direct sequences in the text. For English, this is fine, since word order is fixed (as we have discussed), and grammatical constituents like subjects and verbs are usually close together. As a result, there are enough word n-grams that frequently occur to capture most of the variation. The fact that NLP was first developed for English and its strict word order might explain the enduring success of n-grams.

However, as we have seen, word order in many languages is much freer than it is in English. That means that subjects and verbs can be separated by many more words, and occur at different positions in the sentence. For example, in German,

> "Sie sagte, dass [der Koch]$_{nominative}$ [dem Gärtner]$_{dative}$ [die Bienen]$_{accusative}$ gegeben hatte"

(*She said that [the chef]$_{nominative}$ had given [the bees]$_{accusative}$ to [the gardener]$_{dative}$*, noun cases marked with subscript) could also be written as:

> "Sie sagte, dass [dem Gärtner]$_{dative}$ [die Bienen]$_{accusative}$ [der Koch]$_{nominative}$ gegeben hatte"
>
> "Sie sagte, dass [die Bienen]$_{accusative}$ [dem Gärtner]$_{dative}$ [der Koch]$_{nominative}$ gegeben hatte"

and various other orderings. Not all of them will sound equally natural to a German speaker (they might seem grammatically **marked**). Still, they will all be grammatically valid and possible. In those cases, it is hard to collect sufficient statistics via n-grams, which means that similar documents might look very different as representations. Instead, we would want to collect statistics/counts over the pairs of words that are *grammatically* connected, no matter where they are in the sentence.

marked

In Python, we can use the dependency parser from `spacy` to extract words and their grammatical parents as features, as we have seen before. We simply extract a string of space-separated word pairs (joined by an underscore), and then pass that string to the `CountVectorizer`, which treats each of these pairs as a word:

```
from sklearn.feature_extraction.text import CountVectorizer

features = [' '.join(["{}_{}".format(c.lemma_, c.head.lemma_) for c
    in nlp(s.text)])
 for s in nlp(documents).sents]

vectorizer = CountVectorizer()
X = vectorizer.fit_transform(features)
```

CODE 14. Extracting syntactic feature count representations.

For example, the counts for the first sentence in our examples ("I've been 2 times to New York in 2011, but did not have the constitution for it") would be

```
{'have_be': 2,
'-PRON-_be': 1,
```

```
'be_be': 1,
'2_be': 1,
'time_2': 1,
'to_2': 1,
'new_york': 1,
'york_to': 1,
'in_be': 1,
'2011_in': 1,
',_be': 1,
'but_be': 1,
'do_have': 1,
'not_have': 1,
'the_constitution': 1,
'constitution_have': 1,
'for_have': 1,
'-PRON-_for': 1,
'._be': 1}
```

3.2.3. *TF-IDF Counts.* Count vectors are a good start, but they give equal weight to all terms with the same frequency. However, some terms are just more "interesting" than others, even if they have the same or fewer occurrences. Let us say we have 500 companies, and for each of them several years' worth of annual reports. The collected reports of each company are one document. We will not be surprised to find the words "report" or "company" among the most frequent terms, since they are bound to be used in every document. However, if only some of those companies use "sustainability", but do so consistently in every report, this would be a very interesting signal worth discovering. How could we capture this notion?

When we first look at a corpus, we usually want to know what it is generally about, and what terms we might want to pay special attention to. Frequency alone is not sufficient to find important words: many of the stopwords we purposely remove are extremely frequent. However, there is something about word frequency that carries information. It is just not the raw frequency of each word.

term frequency
document frequency

The answer is to weigh the raw frequency (its **term frequency**, or TF) by the number of documents the word occurs in (its **document frequency**, or DF). This way, words that occur frequently, but in every document (such as function words), will receive very low scores. So will words that are rare, but could occur everywhere (say, "president" in our example: it is not frequent, but will be mentioned by every report). Words that are moderately frequent but occur in only a few documents get the highest score, and these words are what we are after. We can get both of these quantities from a discrete count matrix. The TF is the sum over the corresponding column of the matrix, while the DF is the count of the non-zero entries in that column (see Figure 5).

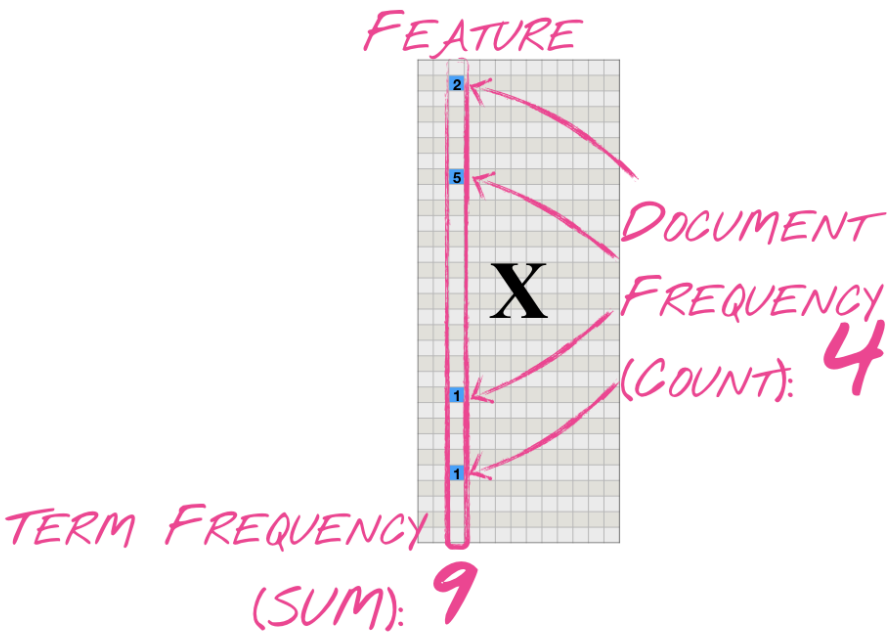


FIGURE 5. Extracting term and document frequency from a count matrix

There are many ways to compute the weighting factor. In the most common one, we check how many documents a term occurs in, and take its **inverse**, or the fraction of all documents in our corpus that contain the word. (This quantity can be obtained by counting the non-zero rows in the respective column of our count matrix.)

inverse

$$IDF(w) = log\frac{N}{DF(w)}$$

This is—quite sensibly–called the **inverse document frequency** (or IDF). Because we often have large corpora and terms occur in only a few, the value of the IDF can get very small, risking an underflow. To prevent this, we take the logarithm.[5] The idea was first introduced by Spärck Jones (1972).

inverse document frequency

By multiplying the two terms together, we get the TF-IDF score of a word:

$$TFIDF(w) = TF(w) \cdot IDF(w)$$

There are several variants to compute both TF and IDF. We can use Laplace smoothing, i.e., we add 1 to each count, to prevent division by 0 (in case a term is not in the corpus). Another possibility is to take the logarithm of the frequency to dampen frequency effects. TF can be computed as a binary measure (i.e., the word is present in the

[5] You might also see the IDF as $-log\frac{DF(w)}{N}$, which comes out to the same.

document or not), as the raw count of the word, the relative count (divided by the length of each document), or a smoothed version:

$$TF_{smoothed} = log(TF(w) + 1)$$

A smoothed version of the IDF is

$$\log \frac{N}{DF(w) + 1} + 1$$

A common variant that implements smoothing and log discounting is

$$TFIDF(w) = (\log TF(w) + 1) \cdot \frac{1}{\log(1 + \frac{N}{DF(w)})}$$

By plotting the IDF versus the TFIDF values, and scaling by the TF, we can see how the non-stopwords in a corpus are distributed. Figure 6 shows the content words in Moby Dick. The five words with the highest TFIDF scores are labeled. While the prevalence of "whale" is hardly surprising (the TFIDF is dominated by the high TF, but then, it's a book about whales), we can see a couple of interesting entries to the right, including the doomed protagonist Ahab, and the archaic form "ye" (for "you").

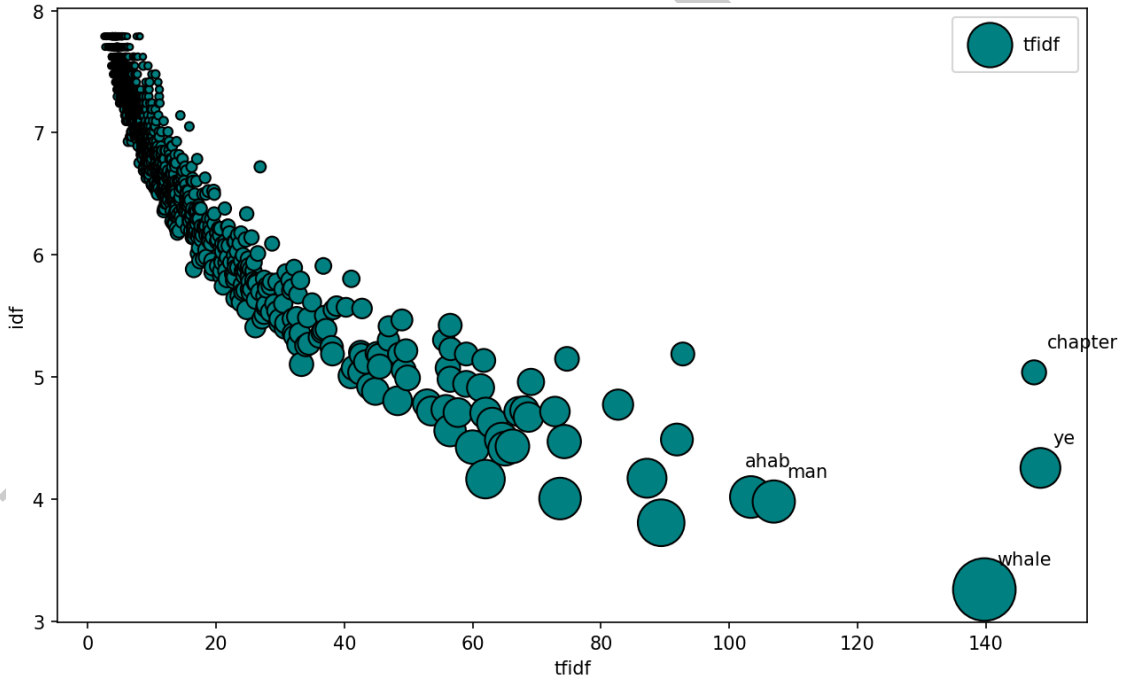


FIGURE 6. Scatter plot of TFIDF and IDF values, scaled by TF, for content words in Moby Dick

In Python, we can use the `TfidfVectorizer` in `sklearn`, which functions very much like the `CountVectorizer` we have already seen:

```
from sklearn.feature_extraction.text import TfidfVectorizer

tfidf_vectorizer = TfidfVectorizer(analyzer='word',
                           min_df=0.001,
                           max_df=0.75,
                           stop_words='english',
                           sublinear_tf=True)

X = tfidf_vectorizer.fit_transform(documents)
```

CODE 15. Extracting TFIDF values from a corpus.

This returns a data matrix with a row for each document, and the transformed values in the cells. We can get the IDF values from the `idf_` property of the vectorizer. In order to get the TFIDF values of a certain feature for the entire corpus, we can sum over the all rows/documents. The `A1` property turns the sparse matrix into a regular array:

```
idf = tfidf_vectorizer.idf_
tfidf = X.sum(axis=0).A1
```

CODE 16. computing IDF and TFIDF values for features.

3.2.4. *Dictionary Methods.* In many cases, we already have prior knowledge about the most important words in a domain, and can use those as features. This knowledge usually comes in the form of word lists, codebooks, or dictionaries, mapping from a word to the associated categories. Several sources and tools exist for applying such **dictionary methods** to text, especially in the field of psychology, such as LIWC (Pennebaker et al., 2001). They come with additional benefits, such as the functionality of scoring texts along several dimensions, such as emotions, psychological traits, etc.

dictionary methods

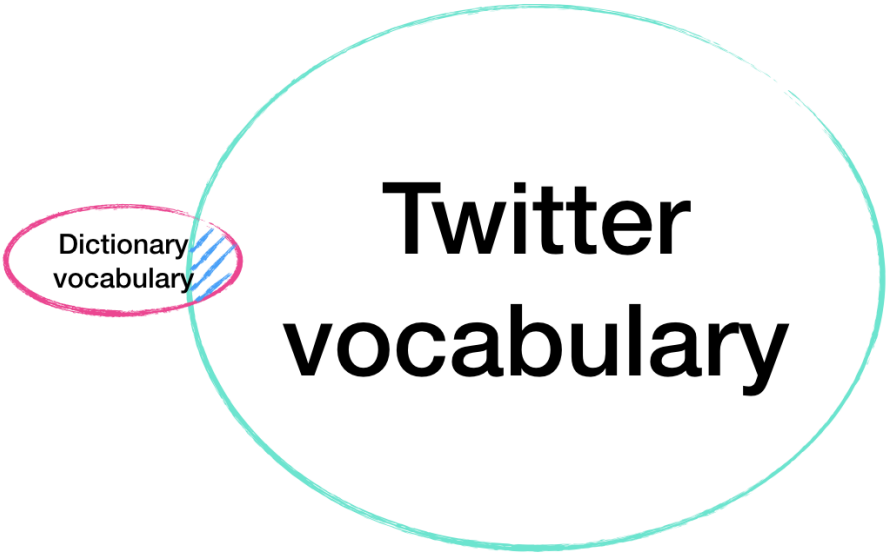


FIGURE 7. Vocabulary overlap between dictionary-based methods and target domain.

While it can be handy to use such a dictionary to get started, it is essential to note that dictionary-based methods have inherent limitations. Their vocabulary and the vocabulary we find in the data will only partially overlap (see Figure 7). In practice, this means that whatever *is* in the dictionary is a perfect candidate (we have high precision in our selection). However, we might miss out on many indicative words because they were not included in the dictionary (our selection has a low recall of all the possible features). Given how flexible language is, it is challenging to cover all possible expressions in a dictionary (Schwartz et al., 2013).

A handy approach is expanding a dictionary-based method with some of the methods discussed here, to expand the dictionary. You can, for example, use the Wordify tool (`https://wordify.unibocconi.it`) to find more terms related to the dictionary categories. This approach will give us bona fide features and will pick up data-specific features. We can substantially extend our initial word list and "customize" it to the data at hand by choosing automatically derived features that strongly correlate with the dictionary terms.

3.3. Distributed Representations. Many times, we would like to capture the meaning similarity between words, such as for **synonyms** like "flat" and "apartment". We might also be interested in their general relatedness in semantic terms. E.g., "doctor," "syringe," and "hospital" all tend to appear in the same context. For example, to capture language change, we might be interested in capturing the most similar words to "mouse" over time (Kulkarni et al., 2015; Hamilton et al., 2016). Or we might want

synonyms

to measure stereotypes by seeing how similar to each other "nurse" is with "man" and "woman", respectively (Garg et al., 2018).

This semantic relatedness of context (called **distributional semantics**) was recognized early on in linguistics by Firth (1957), who stated that

distributional semantics

> "You shall know a word by the company it keeps."

So far, we have treated each word as a separate entity, i.e., "expenditure" and "expenditures" are treated as two different things. We have seen that this is why preprocessing is so important (in this case, lemmatization) because it solves this problem. However, preprocessing does not adequately capture how similar a word is to other words. Distributed representations allow us to do all of these things, by projecting words (or documents) into a high-dimensional space where their relative position and proximity to each other mean something. I.e., words that are similar to each other should be close together in that space. In discrete representations, we defined the dimensions/features, hoping that they capture semantic similarity. In distributed representations, we use contextual co-occurrence as the criterion to arrange words in the given space. Similarity is now an emerging property of proximity in the space. First, though, we need to be more specific by what we mean with *similar to each other*.

3.3.1. *Cosine Similarity.* When we are searching for "flats in milan" online, we will often get results that mention "apartments in milan" or "housing in milan". We have not specified these two other words, but they are obviously useful and relevant. We often get results that express the same thing, but in different words. Intuitively, these words are more similar to each other than to, say, the word "platypus". If we want to find the nearest neighbors, we need a way to express this intuition of semantic similarity between words in a more mathematical way.

For all of the above, the key question is: what does it mean for words to be "similar to each other". Remember that our words are vectors. A vector is nothing but a point specified in the embedding space. So in distributed representations, our words are points in a D-dimensional space. We can draw a line from the **origin** (i.e., the 0 coordinates) of the space to each point. This property allows us to compute the angle between two words, using the **cosine similarity** of two vectors. The angle is a measure of the distance between the two vectors.

origin

cosine similarity

To get the cosine similarity, we first take the dot product between the two vectors, and normalize it by the norm of the two vectors:

$$\cos(A, B) = \frac{A \cdot B}{\|A\|\|B\|} = \frac{\sum_{i=1}^{D} a_i b_i}{\sqrt{\sum_{i=1}^{D} a_i^2}\sqrt{\sum_{i=1}^{D} b_i^2}}$$

The dot product is simply the sum of all pairs of elements multiplied together, and the norm is the square root of the sum over all squared elements. The reason we first square each element and then compute the root is that this gets rid of negative values. So when we sum the numbers up, we end up with a positive number.

Cosine similarity gives us a number between -1.0 and 1.0. A negative value means that the vectors point in diametrally opposite directions. In contrast, a value of 1.0 implies that the two vectors are the same. A value close to 0.0 means the two vectors are perpendicular to each other, or uncorrelated with each other. It does *not* say that the words mean the opposite of each other!

nearest neighbors

We can use cosine similarity to compute the **nearest neighbors** of a word, i.e., to find out which words are closest to a target word in the embedding space. We do this by sorting the vectors of all other words by their cosine similarity with the target. Ideally, we would want the cosine similarity between words with similar meaning ("flat" and "apartment") and between related words ("doctor", "syringe", and "hospital") to be high. The library we will use has a function to do so, but in some cases (e.g., when averaging over several models), we will need to implement it ourselves:

```
import numpy as np
def nearest_neighbors(query_vector, vectors, n=10):
    # compute cosine similarities
    ranks = np.dot(query_vector, vectors.T) / np.sqrt(np.sum(vectors
    **2, 1))
    # sort by similarity, reverse order, and get the top N
    neighbors = [idx for idx in ranks.argsort()[::-1]][:n]
    return neighbors
```

CODE 17. Implementation of nearest neighbors in vector space.

Now we have clarified what it means for words to be similar to each other. The only questions left are: *How do we get all the words into a high-dimensional space? And how do we arrange them by cosine similarity*?

3.3.2. *Word Embeddings.*

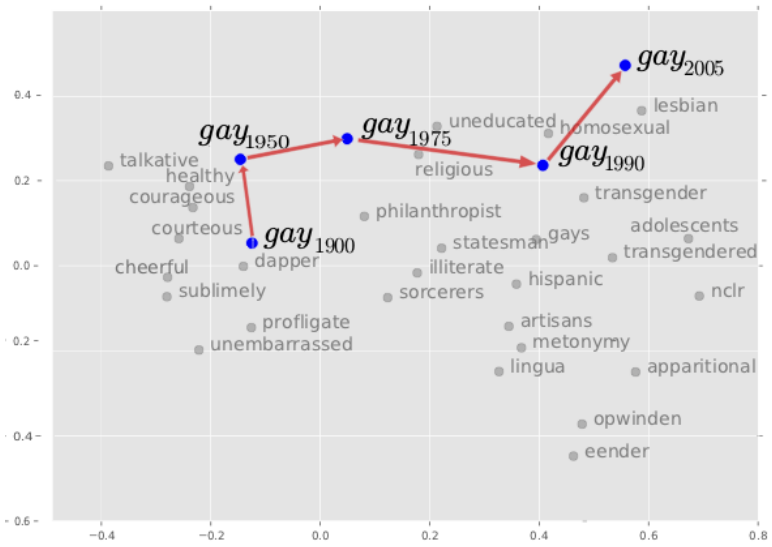


FIGURE 8. Position and change over time of the word "gay" and its nearest neighbors, from Kulkarni et al. (2015)

Let's say we are interested in the changing meaning of words over time (see Figure 8). What did people associate with a word in the early 1900s? Does that differ from its present-day meaning (Kulkarni et al., 2015; Hamilton et al., 2016)? What were people associating with gender roles, ethnic stereotypes, and other psychological biases during different decades (Garg et al., 2018; Bhatia, 2017)? A distributed representation of words from different times allows us to discover these relations by finding the nearest neighbors to a word under different settings.

Following the distributional hypothesis, for words to be similar to each other, they should occur in similar contexts. I.e., if we often see the same two words within a few steps from each other, we can assume that they are **semantically related**. For example, "doctor", and "hospital" will often occur together in the same documents, sometimes within the same sentence, and are therefore semantically similar. If we see two words in the same slot of a sentence, i.e., if we see them interchangeably in the same context, we can assume they are **semantically similar**. For example, look at the slots in the sentences "I only like the ___ gummy bears" and "There was a vase with ___ flowers". Both can be filled with "red" or "yellow" (or "blue", or...), indicating that colors are syntactically similar. Unfortunately, the same is true for **antonyms**. "I had a ___ weekend" can be completed by a wide range of adjectives, but that does not mean that "boring" and "amazing" mean similar things.

semantically related

semantically similar

antonyms: *words that mean the opposite*

Traditionally, distributed representations were collected by defining several **context words**. For each target word in the vocabulary, we would then count how often it occurred in the same document as each of the context words. This approach is known as the **vector space model**. The result was essentially a discrete count matrix. The counts

context words

vector space model

were usually weighted by TF-IDF, to bring out latent similarities, and then transformed by keeping only the k highest eigenvalues from Singular Value Decomposition. The method to produce these dense representations is called **Latent Semantic Analysis** or LSA (Deerwester et al., 1990). Distributional semantic models like LSA used the entire document as context. I.e., we would count the co-occurrence of the target with context words anywhere in the entire text. In contrast, modern embedding models define context as a local window to either side of the target word.

Latent Semantic Analysis

word2vec

Some years ago, a new tool, **word2vec** (Mikolov et al., 2013), made headlines for being fast and efficient in projecting words into vector space, based on a large enough corpus. Rather than relying on a predefined set of context words to represent our target word, this algorithm uses the natural co-occurrence of all words to compute the vectors. (Co-occurrence means the words occur within a particular window size, rather than the whole document.) We only specify the number of dimensions we want to have in our embedding space, and press go.

The intuitive explanation of how the algorithm works is simple. You can imagine the process similar to arranging a bunch of word magnets on a giant fridge. Our goal is to arrange the word magnets so that proximity matches similarity. I.e., words in similar contexts are close to each other, and dissimilar words are far from each other. We initially place all words randomly on the fridge. We then look at word pairs: if we have seen them in the same sentence somewhere, we move them closer together. If they do not occur in the same sentence, we increase the distance. Naturally, as we adjust a word to be close to one of its neighbors, we move it further away from other neighbors. So we need to iterate a couple of times over our corpus before we are satisfied with the configuration.

On a fridge, we only have two dimensions in which to arrange the words, which gives us fewer options for sensible configurations. In `word2vec`, we can choose a higher number, which makes the task easier, as we have many more degrees of freedom to find the right distances between all words. Initial proposals included 50 or 100 dimensions, but for some unknown reason, 300 dimensions seem to work well for most purposes (Landauer and Dumais, 1997; Lau and Baldwin, 2016).

There are two main architectures to implement this intuitive "moving magnets on a fridge" and updating the resulting positions of the vectors: CBOW and the skipgram model. The algorithm iterates over each document in the corpus, and carries out a prediction task. It selects an input word w and a second, output word c from the vocabulary. It then tries to predict the output from the input. In practice, we can either predict the output word from the input of context words (this is called the **continuous bag of words (CBOW) model**, see Figure 9), or predict the context word output from the target word input, which is called **skipgram model** (see Figure 10). Skipgram is somewhat slower, but considered more accurate for low-frequency words.

continuous bag of words (CBOW) model

skipgram model

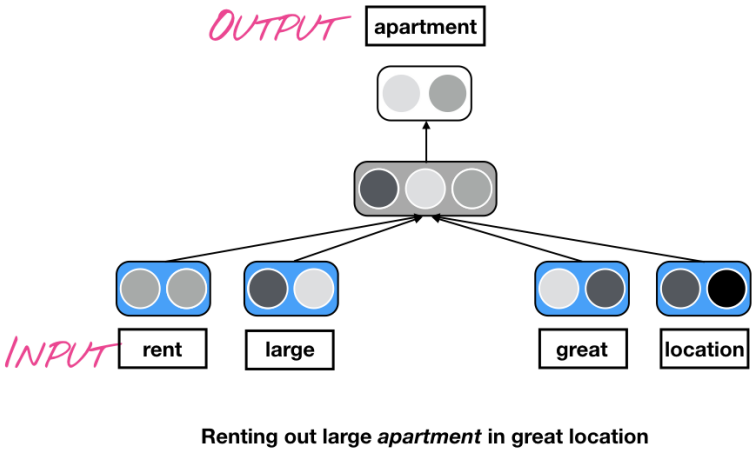


FIGURE 9. Schematic of `Word2vec` model with CBOW architecture for 5 words with two dimensions. Context word vectors in blue, target word vector in white.

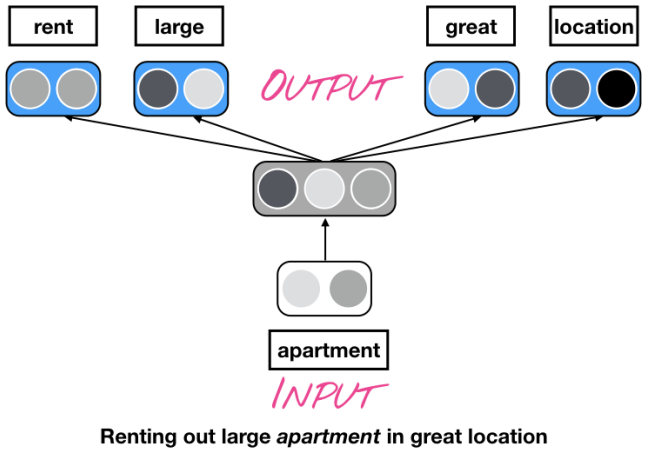


FIGURE 10. Schematic of `Word2vec` model with skipgram architecture for 5 words with two dimensions. Context word vectors in blue, target word vector in white.

In either case, we keep separate matrices for the target and context versions of each word, and discard the output matrix in the end. I.e., in the CBOW model, we delete the target word matrix, in the skipgram model, we remove the context word matrix. For more details on this, see the explanations by Rong (2014) or Goldberg and Levy (2014).

negative sampling
hierarchical softmax

To update the output word matrices, we need to consider the whole vocabulary, which can be quite large (up to hundreds of thousands of words). There are two algorithms to make this efficient: **negative sampling** and **hierarchical softmax**. Both help us update the vast number of words in our vocabulary in an efficient manner.

In negative sampling, the model chooses a random sample of context words c. These words are labeled either 1 or -1, depending on whether they occur within a fixed window either side of the target word w. The model computes the dot product of the input and output vectors and checks the sign of the result. If the prediction is correct, i.e., the sign is positive, and c does indeed occur in the context of w, or if the sign is negative and v does indeed *not* occur in the context of w, nothing happens. If the prediction is incorrect, i.e., the predicted sign differs from the label, the model updates the selected vectors. If we iterate this process often enough, words that occur within a certain window size of each other get drawn closer and closer together in the vector space.

softmax

Given an input word, we can check for a few sampled words, whether they occur in that word's context. However, we can instead also predict the output probability of *all* vocabulary words. Then we choose the word with the highest probability as the prediction. This probability distribution is called a **softmax**. Unfortunately, though, our vocabulary is usually quite large. Computing the softmax for every input word over all output words involves an exponential number of comparisons. The time complexity of this computation becomes quickly prohibitive. The hierarchical softmax addresses this problem. We build a tree on top of our vocabulary, starting with each vocabulary word as a leaf node. We then combine leaf nodes to form higher layers of nodes until we have a single root node. We can use this tree to get the prediction probabilities, and only need to update the paths through this tree to compute the probabilities. This structure cuts the number of necessary computations by a large factor (the logarithm of the vocabulary size). For a much more in-depth explanation and derivation of the methods, see Rong (2014).

The quality of the resulting word vectors depends – as always – on the quality of the input data. One common form of improving quality is to throw out stop words or, rather, to keep only content words. We can achieve this through POS information. Adding POS also disambiguates the different uses of homonyms, e.g., "show_NOUN" vs. "show_VERB".

There are many pre-trained word embeddings available, such as GloVe (Pennington et al., 2014) or Fasttext (Grave et al., 2018), among others. They are based on vast corpora, and in some cases, embeddings exist for several languages in the same embeddings space, making it possible to compare across languages. If we are interested in general societal attitudes, or if we have tiny data sets, these **pre-trained embeddings** are a great starting point.

pre-trained embeddings

In other cases, we might want to train embeddings on our corpus ourselves, to capture the similarities which are specific to this data collection. In the latter case, we can use the `word2vec` implementation in the `gensim` library. The model takes a list of list of strings as input.

```
from gensim.models import Word2Vec
from gensim.models.word2vec import FAST_VERSION

# initialize model
w2v_model = Word2Vec(size=300,
                     window=5,
                     hs=0,
                     sample=0.000001,
                     negative=5,
                     min_count=10,
                     workers=-1,
                     iter=5000
)

w2v_model.build_vocab(corpus)

w2v_model.train(corpus, total_examples=w2v_model.corpus_count, epochs
    =w2v_model.epochs)
```

CODE 18. Initializing and training a `word2vec` model.

We need to pay attention to the parameter settings. The **window size** determines whether we pick up more on semantic or syntactic similarity. The **negative sampling rate** and the number of negative samples affect how much frequent words influence the overall result (`hs=0` tells the model to use negative sampling). As before, we can specify a minimum number of times a word needs to occur to be embedded. The number of threads on which to train (`workers=-1` tells the model to use all available cores to spawn off threads, which speeds up training). The number of iterations controls how often we adjust the words in the embeddings space.

window size

negative sampling rate

Note that there is no guarantee that we find the best possible arrangement of words, since word embeddings start randomly and are changed piecemeal. Because of this stochastic property, it is recommended to train several embedding models on the same data, and then average the resulting embedding vectors for each word over all of the models (Antoniak and Mimno, 2018).

We can access the trained word embeddings via the `wv` property of the model.

```
word1 = "clean"
word2 = "dirty"

# retrive the actual vector
w2v_model.wv[word1]

# compare
w2v_model.wv.similarity(word1, word2)

# get the 3 most similar words
w2v_model.wv.most_similar(word1, topn=3)
```

CODE 19. Retrieving word vector information in the trained `Word2vec` model.

vector space semantics

The resulting word vectors have several neat semantic properties (similarity of meaning), which allow us to do **vector space semantics**. The most famous example of this is the gender analogy of $king - man + woman = queen$. If we add the vectors of "king" and "woman" and then subtract the vector of "man", we get a vector that is quite close to the vector of the word "queen." (In practice, we need to find the nearest actual word vector of the resulting vector.) Other examples show the relationship between countries and their capitals, or the most typical food of countries. These tasks are known as **relational similarity prediction**.

relational similarity prediction

```
# get the analogous word
w2v_model.wv.most_similar(positive=['king', 'woman'], negative=['man'
    ]
```

CODE 20. Doing vector semantics with the trained `word2vec` model.

3.3.3. *Document Embeddings.* Imagine we want to track the language variation in a country city by city: how does the language in A differ from that in B? We have data from each city, and would like to represent the cities on a continuous scale (dialects in any language typically change smoothly, rather than abruptly). We can think of document embeddings as a linguistic profile of a document, so we could achieve our goal if we manage to project each city into a higher dimensional space based on the words used there.[6]

Often, we are not interested in representing individual words, but in representing

[6]This is precisely the idea behind the paper by Hovy and Purschke (2018), who find that the resulting city representations not only capture the dialect continuum (which looks nice on a map), but that they are also close in embedding space to local terms, and can be used in prediction tasks.

an entire document. Previously, we have used discrete feature vectors in a BOW approach to do so. However, BOW vectors only count how often each word was present in a document, no matter where they occur. The equivalent in continuous representations is taking the aggregate over the word embeddings in a document. We concatenate the word vectors row-wise into a matrix D, and compute either the mean or the sum over all rows:

$$e(D) = \sum_{w \in D} e(w)$$

or

$$e(D) = \frac{\sum_{w \in D} e(w)}{|D|}$$

However, we can also use a similar algorithm to what we have used before to learn separate **document embeddings**. (Le and Mikolov, 2014) introduced a model similar to `word2vec` to achieve this goal. The original paper called the model **paragraph2vec**, but it is now often referred to with the name of the successful `gensim` implementation, **`doc2vec`**.

document embeddings

paragraph2vec

`doc2vec`

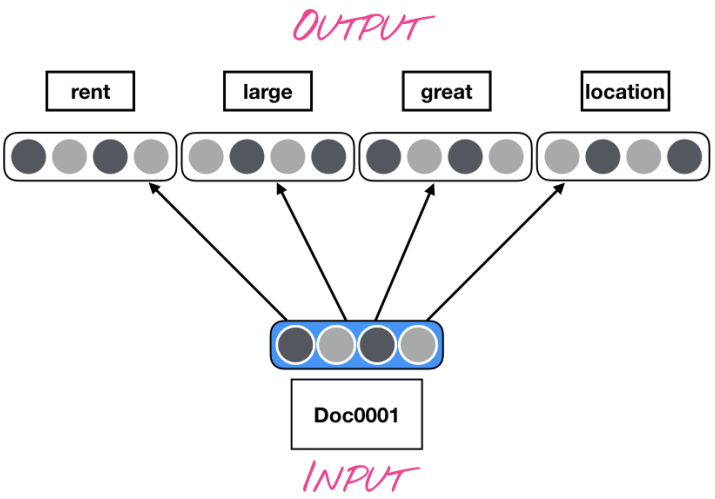


FIGURE 11. `Doc2Vec` model with PV-DBOW architecture

Initially, we represent each word and each document randomly in the same high-dimensional vector space, albeit with separate matrices for the documents and words. We can now either predict target words based on a combined input of their context plus the document vector (similar to the CBOW of `word2vec`). This model architecture is called the Distributed Memory version of Paragraph Vector (PV-DM). Alternatively, we can predict context words based solely on the document vectors (this

is called Distributed Bag of Words version of Paragraph Vector, or PV-DBOW, see Figure 11). The latter is much faster in practice and more memory-efficient, but the former structure is considered more accurate.

The Python implementation in `gensim`, `doc2vec`, is almost identical to the `word2vec` implementation. The main difference is that the input documents need to be represented as a special object called `TaggedDocument`, which has separate entries for `words` and `tags` (the document labels). Note that we can use more than one label! If we do not know much about our documents, we can simply give each of them a unique ID (in the code below, we use a 4-digit number with leading zeroes, i.e., `0002` or `9324`). However, if we know more about the documents (for example, the political party who wrote them, or the city they originated from, or their general sentiment), we can use this.

```
from gensim.models import Doc2Vec
from gensim.models.doc2vec import FAST_VERSION
from gensim.models.doc2vec import TaggedDocument

corpus = []
for docid, document in enumerate(documents):
    corpus.append(TaggedDocument(document.split(), tags=["{0:0>4}".
    format(docid)]))

d2v_model = Doc2Vec(size=300,
                    window=5,
                    hs=0,
                    sample=0.000001,
                    negative=5,
                    min_count=10,
                    workers=-1,
                    iter=5000,
                    dm=0,
                    dbow_words=1)

d2v_model.build_vocab(corpus)

d2v_model.train(corpus, total_examples=d2v_model.corpus_count, epochs
    =d2v_model.epochs)
```

CODE 21. Training a `Doc2vec` model.

The parameters are the same as for `word2vec`, with the addition of `dm`, which decides the model architecture (1 for distributed memory (PV-DM), 0 for distributed bag of words (PV-DBOW)), and `dbow_words`, which controls whether the word vectors are trained (1) or not (0). Not training the word vectors is faster. `doc2vec` stores separate

embedding matrices for words and documents, which we can access via `wv` (for the word vectors) and `docvecs`.

```
target_doc = '0002'

# retrieve most similar documents
d2v_model.docvecs.most_similar(target_doc, topn=5)
```

In general, `doc2vec` provides almost the same functions as `word2vec`.

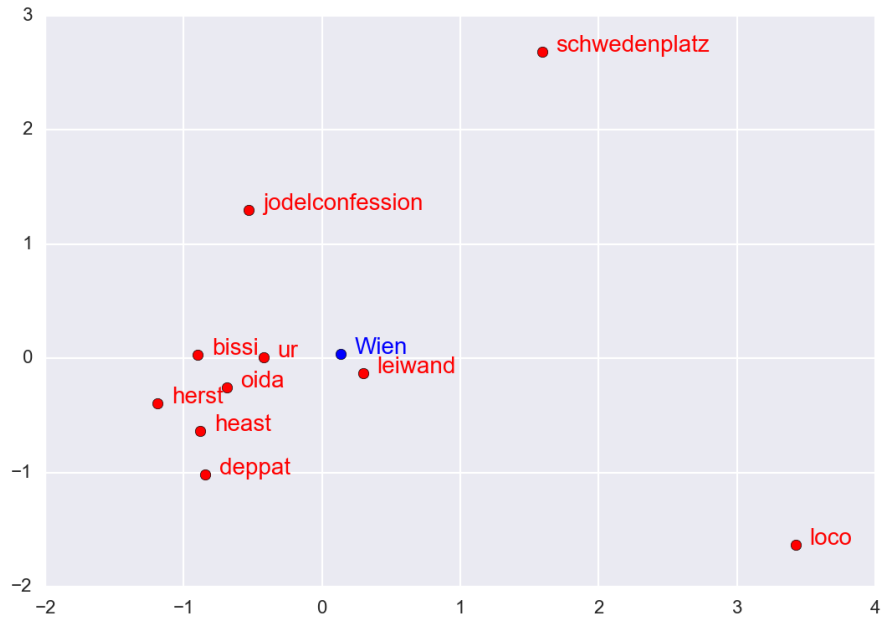


FIGURE 12. Example of documents (cities) and words in the same embedding space from Hovy and Purschke (2018). The words closest to the city vector are dialect terms.

Projecting both word and document embeddings into the same space allows us to compare not just words to each other, which can tell us about biases and conceptualizations. It also allows us to compare words to documents, which can tell us what a specific document is about (see example in Figure 12). And of course, we can compare documents to documents, which can tell us which documents are talking about similar things. For all of these comparisons, we use the nearest neighbor algorithm (see Code 17).

3.4. Discrete vs. Continuous. Which representation we choose depends on several things, not least the application.

Discrete representations are interpretable, i.e., they can assist us in finding a (causal) explanation of the phenomenon we investigate. The dimensions are either predefined by

us, or based on the (most frequent or informative) words in the corpus. Each dimension means something, but many of them will be empty. A discrete data matrix is a bit like a Swiss cheese.

Continuous representations are only interpretable as a whole. The central criterion is similarity (as measure by distance), the individual dimensions have no meaning any more – they are chosen by us, but filled by the algorithm. Almost every cell will be non-zero, so a continuous data matrix is more like a brie than a block of Swiss cheese.

rationalism
empiricism

The difference between representations is also similar to the intended use, mirrored in two schools of thought: **rationalism** and **empiricism**.[7] In the first, we would like a model that helps us *understand* how the world works, and make sure the model is *interpretable*. In the second case, we want a model that *performs* well on a task, even if we do not understand how it does so. Ultimately, though, we need both approaches to make scientific progress.

[7] See also Peter Norvig's excellent blog post about the difference in scientific approaches to machine learning: `http://www.norvig.com/chomsky.html`

Now that we have seen what the basic building blocks of linguistic analysis are, we can start looking at extracting information out of the text. We will see how to search for flexible patterns in the data to identify things like email addresses, company names, or numerical amounts in section 4. Then, we will look at how we can qauntify the information in language, by using probability distributions and entropy (section 5). In section 3.2.3, we will see how to find the most meaningful and important words in a corpus (given that pure frequency is not a perfect indicator). And in section 6, we will see how to capture that some words occur much more frequently together than on their own, and how to define this notion.

4. Regular Expressions

Say you are working with a large sample of employment records from Italian employees, with detailed information about when they worked at which company, and a description of what they did there. However, the names of the companies are contained in the text, rather than in a separate entry. You want to extract the names to get a rough overview of how many different companies there are and to group the records accordingly. Unfortunately, the NER tool for Italian does not work as well as you would like. You do know, though, that Italian company names often end in several abbreviations (similar to "Co.", "Ltd.", or "Inc."). There are only two you want to consider for now ("s.p.a." and "s.r.l."), but they come in many different spelling variations (e.g., *SPA*, *spa*, *S.P.A.*, etc.). How can you identify these markers and extract the company names right before them?

In the same project, you have collected survey data from a large sample of employees. Some of them have left their email address as part of the response, and you would like to spot and extract the email addresses. You could look for anything that comes before or after an @-sign, but how do you make sure it is not a Twitter handle? And how do you make sure it is a valid email address?[8]

String variation is a textbook application of **regular expressions** (or RegEx). They are flexible **patterns** that allow you to specify what you are looking for in general, and how it can vary. In order to do so, you simply write the pattern you want to match.

regular expressions
patterns

sequence	matches
`e`	any single occurrence of e
`at`	`at, rat, mat, sat, cat, attack, attention, later`

TABLE 1. Examples of simple matching.

[8]With "valid", we don't mean whether it actually exists (you could only find out by emailing them), but only whether it is correctly formatted (so that you actually *can* email them).

In Python, we can specify regular expressions and then apply them to text with either `search()` or `match()`. The former checks whether a pattern is *contained* in the input, the latter checks whether the pattern *completely matches* the input:

```
import re
pattern = re.compile("at")

re.search(pattern, 'later') # match at position (1,3)
re.match(pattern, 'later') # no match
```

quantifiers

Sometimes, we know that a specific character will be in our pattern, but not how many times. We can use special **quantifiers** to signal that the character right before them occurs any number of times, including zero (`*`), exactly one or zero times (`?`), or one or more times (`+`). See examples in Table 2.

quantifier	means	example	matches
`?`	0 or 1	`fr?og`	`fog, frog`
`*`	0 or more	`cooo*l`	`cool, coool`
`+`	1 or more	`hello+`	`hello, helloo, hellooooooo`

TABLE 2. Examples of quantifiers.

In Python, we simply incorporate them into the patterns:

```
pattern1 = re.compile("fr?og")
pattern2 = re.compile("hello+")
pattern3 = re.compile("cooo*l")
```

REs also provide a whole set of special characters to match certain types of characters or positions (see Table 3).

classes

However, maybe we don't want really *any* old character in a wildcard position, only one of a small number of characters. For this, we can define **classes** of allowed characters.

In Python, we can use character classes to define our patterns (see also the examples in Table 4):

```
pattern = re.compile('[bcr]at')
```

groups

So far, we have only defined single characters and their repetitions. However, RegExes also allow us to specify entire **groups**, as shown in Table 5:

In order to apply REs to strings in Python, we have several options. We can `match()` input strings, which will only succeed if it can cover the entire string. If we are only interested in whether the pattern is contained as a substring, we can use `search()` instead.

character	means	example	matches
.	any single character	.el	eel, Nel, gel
\n	newline character (line break)	\n+	One or more line breaks
\t	a tab stop	\t+	One or more tabs
\d	a single digit [0-9]	B\d	B0, B1, ..., B9
\D	a non-digit	\D.t	' t, But, eat
\w	any alphanumberic character	\w\w\w	Top, WOO, bee,...
\W	non-alphanumberic character		
\s	a whitespace character		
\S	a non-whitespace character		
\	"Escapes" special characters to match them	.+\.com	abc.com, united.com
^	the beginning of the input string	^...	First 3-letter word in line
$	the end of the input string	^\n$	Empty line

TABLE 3. Examples of special characters.

class	means	example	matches
[abc]	Match any of a, b, c	[bcrms]at	bat, cat, rat, mat, sat
[^abc]	Match anything BUT a, b, c	te[^]+s	tens, tests, teens, texts, terrors...
[a-z]	Match any lowercase character	[a-z][a-z]t	act, ant, not, ... wit
[A-Z]	Match any uppercase character	[A-Z]...	Ahab, Brit, In a, ..., York
[0-9]	Match any digit	DIN A[0-9]	DIN A0, DIN A1, ..., DIN A9

TABLE 4. Examples of character classes.

group	means	example	matches
(abc)	Match sequence abc	.(ar).	hard, cart, fare...
(ab|c)	Match ab OR c	(ab|C)ate	abate, Cate

TABLE 5. Examples of groups.

```
1 document = 'the batter won the game'
```

```
matches = re.match(pattern, document) # returns None
searches = re.search(pattern, document) # matches 'bat'
```

Either of these operations return an object that is `None` if the pattern could not be applied. Otherwise, it will provide us with several properties of the results: `span()` gives us a tuple of the substring positions that match the pattern (and which can be used as indices for slicing), while `group()` returns the matched substring. For our example above, `searches.span()` will return `(4, 7)`, while `searches.group()` returns `'bat'`. If we define several groups in our pattern, we can access them via `groups()`, either jointly or individually.

```
word = 'preconstitutionalism'
affixes = re.compile('(...).+(...)')
results = re.search(affixes, word)
```

Here, `results.groups()` will return the prefix and the suffix found in the input, i.e., `('pre', 'ism')`.

Lastly, we can also use the RegEx to replace elements of the input string that match the pattern via `sub()`. We have mentioned earlier how we can replace the digits of any number with a special token, for example `0`. The following code does this:

```
numbers = re.compile('[0-9]')
re.sub(numbers, '0', 'Back in the 90s, when I was a 12-year-old, a CD
    cost just 15,99EUR!')
```

CODE 22. Replacing patterns in a string.

The code in 22 will return

`'Back in the 00s, when I was a 00-year-old, a CD cost just 00,00EUR!'`.

A **warning**: regular expressions are extremely powerful, but you need to know when to stop. A well-written RegEx can save you a lot of work, and accomplish quite a bit in terms of extracting the right bits of text. However, RegExes can quickly become extremely complicated when you add more and more variants that you would like to account for. It's very easy to write a RegEx that captures way too much (false positives), or is so specific that it captures way too little (false negatives). It is therefore always a good idea to write a couple of positive and negative examples, and make sure the RegEx you develop matches what you want.

5. Entropy

In 1948, Claude Shannon published a groundbreaking paper, "A Mathematical Theory of Communication". In it, Shannon proposed that all communication could be boiled down to a simple concept: sending messages from one point to another. This idea may seem trivial, but Shannon's paper was groundbreaking because it showed that

all communication systems, whether telegraphs, telephones, or computers, could be analyzed using the same mathematical framework. The central concept is that of **entropy**: a measure of the amount of information in a message or the uncertainty you have about it. Allegedly, coming up with this insight involved a lot of games of hangman with his wife. How come? **entropy**

Hangman, or more contemporary Wordle, are great examples of entropy. In the most general sense, entropy is a measure of disorder or uncertainty in an information context. The more disorder, the higher the entropy. In the context of guessing a word, the more possible ways to complete a guess to form actual words, the higher the entropy. In Wordle, you have six tries to guess a five-letter word. After each guess, you are told for each character in your guess whether it is not in the target word (grey), in the target word but in a different position (yellow), or in that position in the target word (green). In the beginning, you know nothing about the target word, so the entropy is very high. But note that even before this first guess, you already have some information based on the fact that the target is an English word: any words with the letters S, A, L, E, T, or R are much more likely than any words with Q or X. As you guess more and more letters, the entropy decreases because the number of possible words you can still form decreases. But entropy also helps us to quantify how complicated the solutions are. If you have the pattern _OUND, you could complete it with W, R, P, S, F, H, B, or M to form possible English words. In contrast, for the pattern SOUN_, the only possible English word that completes it is the letter D. The entropy is much higher in the first case than in the second case.

The most important notion here is the probability of a word.[9] One way to think about a word's probability is this: if we opened a very large book and randomly pointed at one word, how often would it be the one we are looking for? In practice, you would count how often each word appears in the huge book and then divide by the total number of words. The result is a (usually very, very small) number between 0 and 1.[10] If the following sentence was all you had, the word "good" appears 2 times in 10 words.

A good word is a word that has good characters

So the probability of "good" is 0.2, or 2 divided by 10. The bigger the book, the more accurate the probability will be (if your book has only one page, you probably won't see all the words in a language, and most of them will only occur once). Today, that book is essentially the entire Internet, so we have a pretty good estimate. However, the numbers are also really, really, really small. We can take their logarithm to make it easier to work with them. For our purpose, the logarithm is a function that makes small numbers larger and larger numbers smaller, so it's a squashing function. There are different types of logarithms, but we use the one with base 2: Shannon wanted to

[9]The famous linguist Noam Chomsky once grumpily declared that this notion made no sense. It might not be in a day-to-day sense, but in a model, it absolutely does, as we will see.

[10]In science, people use these small numbers, e.g., 0.9. In common writing, we would use percentages. To get from probability to percentages, you multiply by 100. So probability 0.9 means a 90% chance.

quantify how many binary bits you would need to encode something. I.e., how many binary switches would you need to turn on and off (i.e., bits) to assign a unique sequence to each word? The logarithms tells us how many bits we would need. In this case, 2.3. We then multiply the two parts, the probability and its logarithm, and because taking the logarithm gives us a negative number, so we also have to take its absolute value. That whole process can be written more succinctly as[11]

$$H(x) = -p(x)log_2p(x)$$

Shannon had previously showed how the on-and-off switches in computers could be used to express logical true or false. With entropy, he could say how many combinations of switches we would need to encode, say, English. Entropy can generally be used to compute how many words/tokens/characters/signs we need to express all possible inputs efficiently. Essentially, entropy measures how much information can be squeezed into a text. By understanding the entropy of a language, we can encode and decode messages more efficiently.

This insight also makes intuitive sense from a linguistic perspective: we want to be able to express a lot of different things, but we don't want to make it too difficult for our friends to understand us. A language with only one word for everything would be as difficult or useless in that respect as a language with a unique word for each and every possible concept.

To summarize, the probability of a word is a measure of how often it appears compared to all other words. We can calculate this probability by counting the number of times it appears in a large sample and then dividing that by the total number of words. The result is a number between 0 and 1, which we can then convert to bits by taking the logarithm. Entropy is highest when all words are equally likely (i.e., when we cannot reasonably guess which comes next) and lowest if only one option exists (because you don't have to guess).

6. Pointwise Mutual Information

Intuitively, we would like to treat terms like "social media", "New York", or "greenhouse emission" as a single concept, rather than treating them as individual units delimited by white space. However, just looking at the frequency of the two words together is not sufficient. The word pair "of the" is extremely frequent, but nobody would argue that this is a single meaningful concept. To complicate matters more, each part of these often also occurs by itself. Think of the elements in "New York" (while "York" is probably not too frequent, "New" definitely is). Leaving these words apart or joining them will have a huge impact on our findings. How do we join the right words together?

Word is a squishy concept, though. We have seen that whitespaces are more of an optional visual effect than a meaningful word boundary (e.g., Chinese does not

[11] Entropy is traditionally denoted with H.

use them). Even the (in)famous German noun compounds ("Nahrungsmittelettikierungsgesetz", or "*law for the labeling of food stuffs*") can be seen as just a more whitespace-efficient way of writing the words.

Even in English, we have concepts that include several words, but that we would not (anymore) analyze as separate components. Think of place names like "Los Angeles" and "San Luis Obispo", but also concepts like "social media". (If you disagree on the latter, that is because the process, called **grammaticalization**, is gradual, and in this case not fully completed.) While it is still apparent what the parts mean, they have become inextricably linked with each other. These word pairs are called **collocations**.

grammaticalization

collocations

When we work with text, it is often better to treat collocations as one entity, rather than as separate units. To do this, we usually just replace the white space between the words with an underscore (i.e., we would use `los_angeles`). There are some "general" collocations (like "New York" or "Los Angeles"). However, nearly every domain includes specialized concepts relevant to the context of our analysis. For example, "quarterly earnings" in earnings reports, or "great service" if we are analyzing online reviews.

We can compute the probability of each word occurring on its own, but this does not help much: "New" is much more likely to occur than "York". We need to relate these quantities to how likely the two words are to occur together. If either (or all) of the candidate words occur in the context of the others, we have reason to believe it is a collocation. E.g., "Angeles" in the case of "Los Angeles". If, on the other hand, either word is just as likely to occur independently, we probably do not have a collocation. E.g., "of" and "the" in "of the."

To put numbers to this intuition, we use the **joint probability** of the entire n-gram occurring in a text and normalize it by the individual probabilities (for more on probabilities, see Appendix 1). Since the probabilities can get fairly small, we typically use the logarithm of the result. So, for a bigram we have

joint probability

$$PMI(x;y) = \log \frac{P(x,y)}{P(x)P(y)}$$

Joint probabilities can be rewritten as conditional probabilities, i.e.,

$$P(x,y) = P(y) \times P(x|y) = P(x) \times P(y|x)$$

The last two parts are the same because we don't care about the order of x and y. So if it's more convenient, we can transform our PMI calculation into

$$PMI(x;y) = \log \frac{P(x|y)}{P(x)}$$

or

$$PMI(x;y) = \log \frac{P(y|x)}{P(y)}$$

Essentially, the latter tells us for a bigram "xy" how likely we are to see x followed by y (i.e., $P(y|x)$), normalized by how likely we are to see y in general ($P(y)$).

Applying collocation detection to an example corpus in Python is much easier, since there are some functions implemented in `nltk` that save us from computing all the probability tables:

```
from nltk.collocations import BigramCollocationFinder,
    BigramAssocMeasures
from nltk.corpus import stopwords

stopwords_ = set(stopwords.words('english'))

words = [word.lower() for document in documents for word in document.
    split()
          if len(word) > 2
          and word not in stopwords_]
finder = BigramCollocationFinder.from_words(words)
bgm = BigramAssocMeasures()
collocations = {bigram: pmi for bigram, pmi in finder.score_ngrams(
    bgm.mi_like)}
collocations
```

CODE 23. Finding collocations with `nltk`.

The code first creates a long list of strings from the corpus, removing all stopwords and words shorter than 2 characters (thereby removing punctuation). It then creates a `BigramCollocationFinder` object, initialized with the word list. From this object, we can extract the statistics. Here, we use a variant of PMI, called `mi_like`, which does not take the logarithm, but uses an exponent.

Table 6 shows the results for the text sample from Moby Dick. We can then set a threshold above which we concatenate the words.

preprocessing

All of the last few sections are examples of **preprocessing**, to get rid of noise and unnecessary variation, and to increase the amount of signal in the data.

All of these steps can be combined, so we get a much more homogenous, less noisy version of our input corpus. However, there is no general rule of what steps to exclude and which to keep. For topic models (see section 4), removing stopwords is crucial. However, for other tasks, such as author attribute prediction, the number and choice of stopwords, such as pronouns, is essential. We might not need numbers for sentiment analysis, so we can either remove them or replace them with a generic number token. However, we would definitely want to keep all numbers if we are interested in tracking earnings.

RANK	COLLOCATION	$MI(x;y)$
1	moby_dick	83.000000
2	sperm_whale	20.002847
3	mrs_hussey	10.562500
4	mast_heads	4.391153
5	sag_harbor	4.000000
6	vinegar_cruet	4.000000
7	try_works	3.794405
8	dough_boy	3.706787
9	white_whale	3.698807
10	caw_caw	3.472222

TABLE 6. MI values and rank for a number of collocations in Moby Dick

Ultimately, which set of preprocessing steps to apply is highly dependent on the task and the goals. It makes sense to try out different combinations on a separate, held-out development set, and observe which one gives the best results. More importantly, preprocessing decisions always change the text, and thereby affect the inferences we can make. We need to be aware of this effect, and we should justify and document all preprocessing steps and decisions we have taken Denny and Spirling (2018).

Part II

Exploration
Finding Structure in the Data

We will now look at ways to visualize the information we have (section 1) and to group texts together into larger clusters (section 2), before we turn to probabilistic models. We discuss how to spot unusual or creative sentences (and potentially generate our own) in section 3; and how to find the most important themes and topics (via topic models) in section 4.

1. Matrix Factorization

Let's say you have run `Doc2Vec` on a corpus and created an embedding for each document. You would like to get a sense of how the documents as a whole relate to each other. Since embeddings are points in space, a graph sounds like the most plausible idea. However, with 300 dimensions, this is a bit hard. How can you reduce the 300 down to 2 or 3? You would also like to visualize the similarity between documents using colors: documents with similar content should receive similar colors. Is there a way to translate meaning into color?

latent dimensions

The data we collect from our count or TFIDF vectorizations is a sample of actual language use. These are the correlations we have observed. However, language is complex and has many **latent dimensions**. Various factors influence how we use words, such as societal norms, communicative goals, broader topics, discourse coherence, and many more. If we think of the word and document representations as mere results of these underlying processes, we can try and reverse-engineer the underlying dimensions.

matrix factorization

Finding these latent factors is what **matrix factorization** tries to do: the methods we will see try to represent the documents and the terms as separate entities, connected through the latent dimensions. We can interpret the latent dimensions as a variety of things, such coordinates in a 2 or 3-dimensional space, RGB color values, or (if we have more than 3) as "topics".

Latent Semantic Analysis *(LSA)*

This view was incredibly popular in NLP in the 1990 and into the early 2000s, thanks to a technique called **Latent Semantic Analysis** (LSA), based on matrix decomposition. The resulting word and document matrices were used for all kinds of semantic similarity computations (between words, between documents), for topics, and for visualization. However, there are now specialized techniques that provide a more concise and flexible solution for everything LSA was used for. Word embeddings (see section 3.3), document embeddings (ibd.), topic models (see section 4), and visualization methods like t-SNE (see below). However, it is still worth understanding the principles behind it, as they can provide a quick and efficient way to analyze data. Matrix factorization is still widely used in **collaborative filtering** for recommender systems. For example, to suggest movies people should watch based on what they and people similar to them have watched before.

collaborative filtering

1.1. Dimensionality Reduction. So far, the dimensionality D of our data matrix X was defined by one of two factors. Either by the size of the vocabulary in our data

(in the case of sparse vectors), or by a pre-specified number of dimensions (in the case of distributed vectors).

However, there are good reasons to reduce this number. The first is that 300 dimensions are incredibly hard to imagine and impossible to visualize. However, to get a sense of what is going on in the data, we often *do* want to plot it. So having a way to project the vectors down to two or three dimensions is very useful for **visualization**.

visualization

The other reason is performance and efficiency: We might suspect that there is a smaller number of **latent dimensions** that is sufficient to explain the variation in the data. Reducing the dimensionality to this number can help us do a better analysis and prediction. The patterns that emerge can even tell us what those latent dimensions were.

latent dimensions

In the following, we will look at two dimensionality-reduction algorithms. We can represent the input data as either sparse BOW vectors or dense embedding vectors. Note that using dimensionality reduction to k factors on a D-dimensional embedding vector is *not* the same as learning a k-dimensional embedding vector in the first place. There is no guarantee that the k embedding dimensions are the same as the latent dimensions we get from dimensionality reduction. Matrix decomposition relies on feature-based input to find a smaller subset. A feature of this approach on document representation matrices is that earlier dimensions are more influential than later ones. At the same time, the embedding-based techniques make use of the entire set of dimensions.

Singular Value Decomposition *(SVD)*

1.1.1. *Singular Value Decomposition.* **Singular Value Decomposition** (SVD) is one of the most well-known matrix factorization techniques. It is an exact decomposition technique based on **eigenvalues**. The goal is to decompose the matrix into three elements:

eigenvalues

$$\mathbf{X}_{n\times m} = \mathbf{U}_{n\times k}\mathbf{S}_{k\times k}(\mathbf{V}_{m\times k})^T$$

where $\mathbf{X}$ is a data matrix of n documents and m terms, $\mathbf{U}$ is a lower-dimensional matrix of n documents and k latent concepts, $\mathbf{S}$ (sometimes also confusingly written with the Greek letter for it, $\mathbf{\Sigma}$, which can look like a summation) is a k-by-k diagonal matrix with non-negative numbers on the diagonal (the principal components, eigenvalues, or "strength" of each latent concept) and 0s everywhere else, and $\mathbf{V}^T$ is a matrix of m terms and k latent concepts.[1]

If we matrix-multiply the elements together, we can get a matrix $\mathbf{X}'$. This matrix has the same number of columns and rows as $\mathbf{X}$, but it is filled with new values, reflecting the latent dimensions. I.e., $\mathbf{X}'$ contains as many features as our chosen number of latent dimensions, and the rest is comprised of zeros.

In Python, SVD is very straightforward:

```
1 from sklearn.decomposition import TruncatedSVD
```

[1]Note that officially, the decomposition is $\mathbf{U}_{n\times\mathbf{n}}\mathbf{S}_{\mathbf{n}\times\mathbf{m}}(\mathbf{V}_{m\times\mathbf{m}})^T$, which is then reduced to k by sorting the eigenvalues and setting all after the kth to 0. However, this is not aligned with the Python implementation.

```
k = 10

svd = TruncatedSVD(n_components=k)
U = svd.fit_transform(X)
S = svd.singular_values_
V = svd.components_
```

CODE 24. SVD decomposition into 10 components.

Principal Component Analysis

SVD is related to another common decomposition approach, called **Principal Component Analysis** (PCA), where we try to find the main dimensions of variation in the data. SVD is one way to do PCA, since the singular values can be used to find the dimensions of variation. PCA can be done in other ways, too, but SVD has the advantage over other methods that it does not require us to have a square matrix (i.e., the same number of rows and columns).

Non-Negative Matrix Factorization

1.1.2. *Non-Negative Matrix Factorization.* **Non-Negative Matrix Factorization** (NMF) fulfills a similar role as SVD. However, it assumes that the observed n-by-m matrix $\mathbf{X}$ is composed of only two matrices, $\mathbf{W}$ and $\mathbf{H}$, which approximate $\mathbf{X}$ when multiplied together. $\mathbf{W}$ is a n-by-k matrix, i.e., it gives us a lower-dimensional view of the documents, and is therefore very similar to what we got from SVD's $\mathbf{U}$. $\mathbf{H}$ is a k-by-m matrix, i.e., it gives us a lower-dimensional view of the terms. It is, therefore, very similar to what we got from SVD's $\mathbf{V}$. The big differences from SVD are that there is no matrix of eigenvalues, nor any negative values. NMF is also not exact, so the two solutions will not be the same.

In Python, NMF is again very straightforward.

```
from sklearn.decomposition import NMF

nmf = NMF(n_components=k, init='nndsvd', random_state=0)
W = nmf.fit_transform(X)
H = nmf.components_
```

CODE 25. Applying NMF to data.

The initialization has a certain impact on the results, and using Nonnegative Double Singular Value Decomposition (NNDSVD) helps with sparse inputs.

1.2. Visualization. Once we have decomposed our data matrix into the lower-dimensional components, we can use them for visualization. If we project down into 2 or 3 dimensions, we can visualize them in 2D or 3D. An example are the clouds in Figure 1: each point represents a song, colored by the genre. The graph immediately shows that there is little to no semantic overlap between Country and Hip-hop. The function for plotting this is given in Code 26 below.

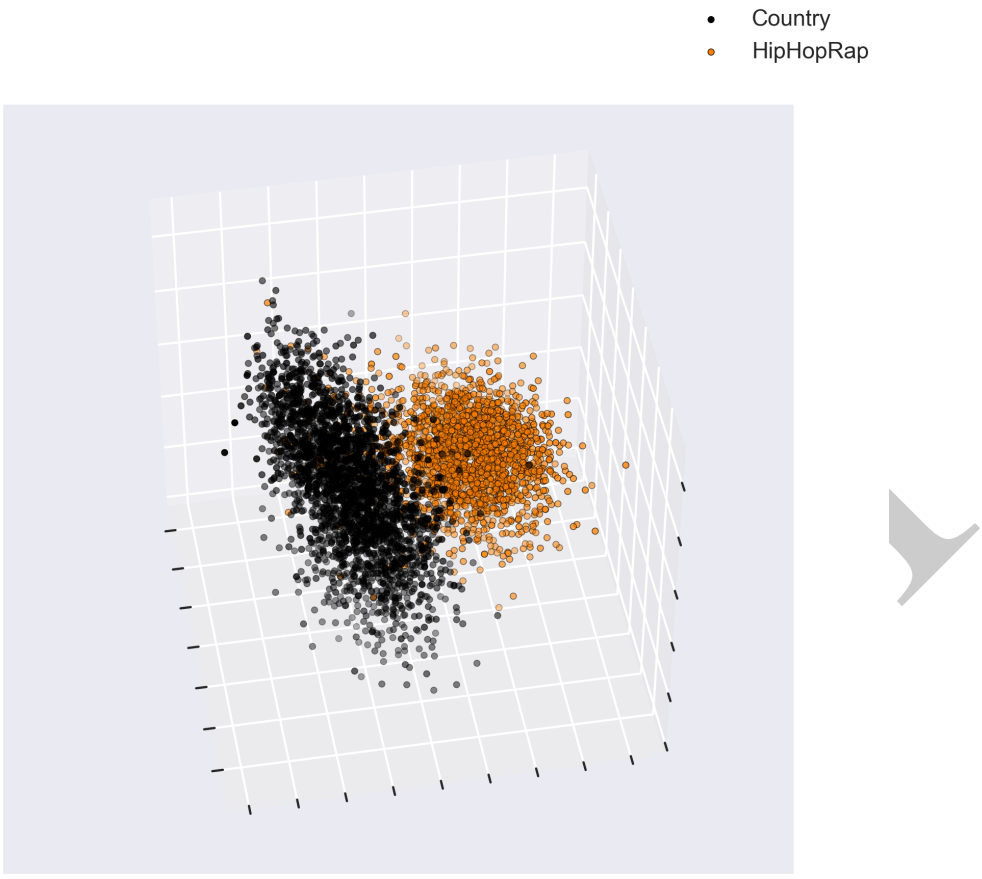


FIGURE 1. Scatter plot of document embeddings for song lyrics in 3D.

```
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
from matplotlib import colors

def plot_vectors(vectors, title='VIZ', labels=None, dimensions=3):
    # set up graph
    fig = plt.figure(figsize=(10,10))

    # create data frame
    df = pd.DataFrame(data={'x':vectors[:,0], 'y': vectors[:,1]})
    # add labels, if supplied
    if labels is not None:
        df['label'] = labels
        print(df.label)
    else:
        df['label'] = [''] * len(df)
```

```
17
18      # assign colors to labels
19      cm = plt.get_cmap('afmhot') # choose the color palette
20      n_labels = len(df.label.unique())
21      label_colors = [cm(1. * i/n_labels) for i in range(n_labels)]
22      cMap = colors.ListedColormap(label_colors)
23
24      # plot in 3 dimensions
25      if dimensions == 3:
26          # add z-axis information
27          df['z'] = vectors[:,2]
28          # define plot
29          ax = fig.add_subplot(111, projection='3d')
30          frame1 = plt.gca()
31          # remove axis ticks
32          frame1.axes.xaxis.set_ticklabels([])
33          frame1.axes.yaxis.set_ticklabels([])
34          frame1.axes.zaxis.set_ticklabels([])
35
36          # plot each label as scatter plot in its own color
37          for l, label in enumerate(df.label.unique()):
38              df2 = df[df.label == label]
39              ax.scatter(df2['x'], df2['y'], df2['z'], c=label_colors[l
    ], cmap=cMap, edgecolor=None, label=label, alpha=0.3, s=100)
40
41      # plot in 2 dimensions
42      elif dimensions == 2:
43          ax = fig.add_subplot(111)
44          frame1 = plt.gca()
45          frame1.axes.xaxis.set_ticklabels([])
46          frame1.axes.yaxis.set_ticklabels([])
47
48          for l, label in enumerate(df.label.unique()):
49              df2 = df[df.label == label]
50              ax.scatter(df2['x'], df2['y'], c=label_colors[l], cmap=
    cMap, edgecolor=None, label=label, alpha=0.3, s=100)
51
52      else:
53          raise NotImplementedError()
54
55      plt.title(title)
56      plt.show()
```

CODE 26. Making scatter plots of vectors in 2D or 3D, colored by label.

We can also use a projection into three dimensions to color our data: by interpreting each dimension as a **color channel** in an RGB scheme, with the first denoting the amount of red, the second the amount of green, and the third the amount of blue. We then can assign a unique color triplet to each document. These triplets can then be translated into a color (e.g., $(0, 0, 0)$ is black, since it contains no fraction of either red, green, or blue). Documents with similar colors can be interpreted as similar to each other. Since the RGB values have to be between 0 and 1, NMF works better for this application, since there will be no negative values. However, the columns still have to be scaled.

color channel

This approach was taken in Hovy and Purschke (2018), where the input matrix consisted of document embeddings of text from different cities. By placing the cities on a map and coloring them with the RGB color corresponding to the 3-dimensional representation of each document vector, they created the map in Figure 2. It accurately shows the dialect gradient between Austria and Germany, with Switzerland different from either of them.

1.2.1. *t-SNE.* While the previous two techniques work reasonably well for visualization, they are essentially by-products of the matrix factorization. There is another dimensionality reduction technique that was specifically developed for the visualization of high-dimensional data. It is called **t-SNE** (t-Distributed Stochastic Neighbor Embedding, Maaten and Hinton (2008)).

t-SNE

The basic intuition behind t-SNE is that we would like to reduce the number of dimensions to something plottable. At the same time, we want to maintain the similarities between neighborhoods of points in the original high-dimensional space. I.e., if two instances were similar to each other in the original space, we want them to be similar to each other in the new space. Also, they should be similar to their respective neighbors in the original space. To achieve this, t-SNE computes the probability distributions over all neighbors for a point in both dimensions. It then tries to minimize the difference between these two distributions. This effort amounts to minimizing the **Kullback-Leibler (KL) divergence**, which measures how different two probability distributions are from each other.

Kullback-Leibler (KL) divergence

In practice, t-SNE can produce much more "coherent" visualizations that respect the inherent structure of the input data. E.g., the widely used MNIST data set is a collection of actual hand-written numbers from letters, which is used to train visual models. We would like to have all examples of hand-written 2s close together, no matter their slant or size of the curve. At the same time, we want to have them separate from all examples of 7s, which in turn should be close to each other. t-SNE can accomplish that. However, the way t-SNE works is stochastic, so no two runs of the algorithm are guaranteed to give us the same result. It also involves several parameters that influence the outcome. It therefore requires some tuning and experimentation before we have a good visualization.

1.3. Word descriptors. In both matrix factorization algorithms we have seen, the input data is divided into at least two elements. First, a n-by-k matrix that can be thought

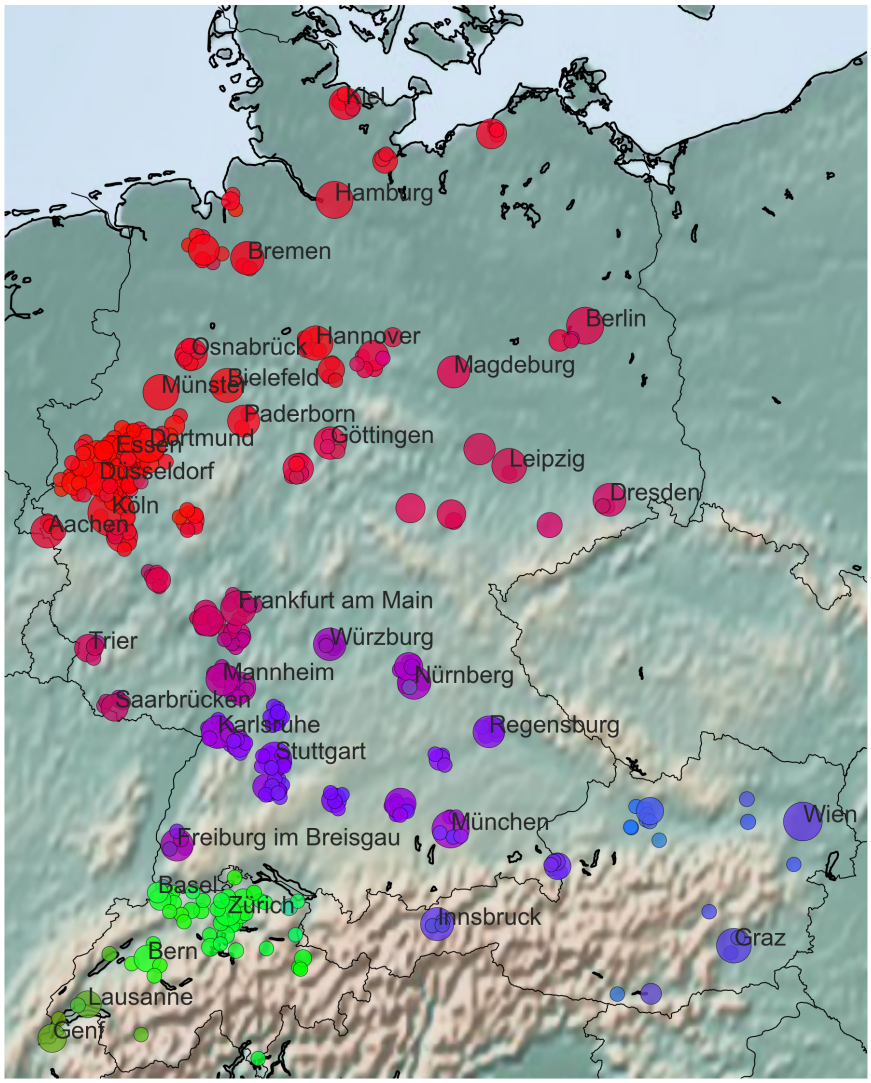


FIGURE 2. Visualization of documents from various German-speaking cities, colored by RGB values (Hovy and Purschke, 2018)

of as the representation of the document representation in lower-dimensional space. In SVD, this was $\mathbf{U}$, in NMF $\mathbf{W}$. Second, there was a k-by-n matrix that can be thought of as the representation of the words in the lower dimensional space, if transposed. However, if we are not transposing it, we can think of it as the affinity of each word with each of the rows of k latent concepts. In SVD, this was $\mathbf{V}$, in NMF $\mathbf{H}$.

We can retrieve the five highest scoring columns for each of the k dimensions. The words corresponding to these five columns "characterize" each of the k latent dimensions. These word lists are similar to a probabilistic topic model. In fact, they were its

precursors. However, again, these lists are only a by-product of the factorization. They are not specifically designed with the structure of language in mind. The decomposition solution determines the topics. If we do not like them, we do not have much flexibility in changing them (other than rerunning the decomposition). Probabilistic topic models, by contrast, offer all kinds of knobs and dials to adjust the outcome. Nonetheless, it is a fast and easy way to get the gist of the latent dimensions, and can help us interpret the factorization.

```
def show_topics(a, vocabulary, topn=5):
    topic_words = ([[vocabulary[i] for i in np.argsort(t)[:-topn
   -1:-1]]
                    for t in a])
    return [', '.join(t) for t in topic_words]
```

CODE 27. Retrieving the word descriptors for latent dimensions.

Here, `a` is the k-by-n matrix, `vocabulary` is the list of words (typically from the vectorizer). For Moby Dick and $k = 10$, this returns

```
['ahab, captain, cried, captain ahab, cried ahab',
 'chapter, folio, octavo, ii, iii',
 'like, ship, sea, time, way',
 'man, old, old man, look, young man',
 'oh, life, starbuck, sweet, god',
 'said, stubb, queequeg, don, starbuck',
 'sir, aye, let, shall, think',
 'thou, thee, thy, st, god',
 'whale, sperm, sperm whale, white, white whale',
 'ye, look, say, ye ye, men']
```

1.4. Comparison. The two algorithms are very similar, with only minor differences. Let's directly compare them side by side. Table 1 compares them on a variety of dimensions.

	SVD	NMF
Negative values (embeddings) as input?	yes	no
number of components	3: **U**, **S**, **V**	2: **W**, **H**
document view?	yes: **U**	yes: **W**
term view?	yes: **V**	yes: **H**
strength ranking?	yes: **S**	no
exact?	yes	no
"topic" quality	mixed	better
sparsity	low	medium

TABLE 1. Comparison of SVD and NMF.

What to ultimately use depends partially on the input and the desired application/output we want, as shown in Table 2.

	Discrete Features	Embeddings
Latent topics	NMF	Not applicable
RGB translation	NMF	SVD + scaling
Plotting	SVD	t-SNE

TABLE 2. Comparison of SVD and NMF.

2. Clustering

Imagine you have a bunch of documents and suspect that they form larger groups, but you are not sure which and how. For example, say you have texts from various cities and suspect that these cities group together in dialect and regiolect areas. How can you test this? And how do you know how good your solution is?[2]

Clustering is an excellent way to find patterns in the data even if we do not know much about it, by grouping documents together based on their similarities. Imagine, for example, that you have self-descriptions of a large number of companies. You are trying to categorize them into a manageable number of industry sectors (technology, biology, health care, etc). If you suspect that the language in the documents reflects these latent properties, you can use clustering to assign the companies to the sectors.

Both clustering and matrix factorization try to find latent dimensions. However, the critical difference is that matrix factorization is reversible (i.e., we can reconstruct the input), while clustering is not.

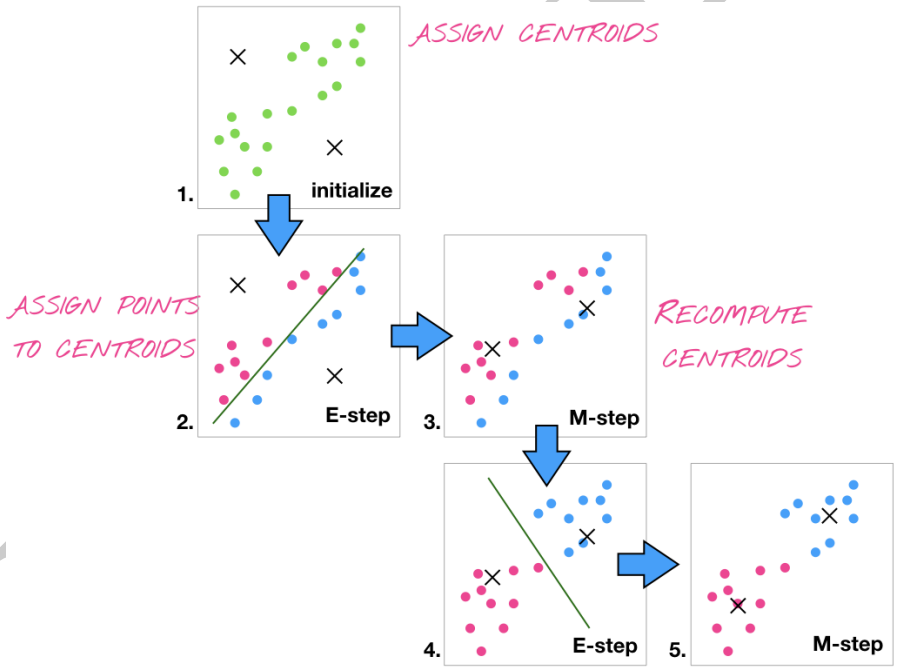


FIGURE 3. k-means clustering.

2.1. K-Means. **K-Means** is one of the simplest clustering algorithms. It chooses k **centroids** of clusters (the means of all the document vectors associated with that

K-Means

centroids

[2]This is exactly the setup in Hovy and Purschke (2018). Spoiler alert: the automatically induced clusters match existing dialect areas to almost 80%.

cluster). It then assigns each document a score according to how similar it is to each mean.

However, we have a bit of a circular problem here: To know the cluster centroids, we need to assign the documents to the closest clusters. But to know which clusters the documents belong to, we need to compute their centroids... This situation is an example of where the EM algorithm (Dempster et al., 1977) comes in handy.

Here is how we can solve it (see Figure 3):

(1) We start by randomly placing cluster centroids on the graph.
(2) Then, we assign each data point to the closest cluster, by computing the distance between the cluster centroids and each point.
(3) Lastly, we compute the centers of those new clusters and move the centroids to that position.

We repeat the last two steps until the centroids stop moving around.

In `sklearn`, clustering with k-means is very straightforward, and it is easy to get both the cluster assignment for each instance as well as the coordinates of the centroids.

```
from sklearn.cluster import KMeans

km = KMeans(n_clusters=10, n_jobs=-1)
clusters = km.fit_predict(X)
centroids = km.cluster_centers_

```

CODE 28. K-means clustering with 10 clusters, using all cores.

K-means is quite fast and scales well to large numbers of instances. However, it has the distinct disadvantage of being stochastic. Because we set the initial centroids at random, we will get slightly different outcomes every time we run the clustering. If we happen to know something about the domain, we can instead use these points as initial cluster centroids. For example, if we know that certain documents are good exemplars of a suspected cluster, we can let the algorithm start from them.

2.2. Agglomerative Clustering. Rather than dividing the space into clusters and assigning data points wholesale, we can also build clusters from the bottom up. We start with each data point in its own cluster and then merge them, again and again, to form ever-larger groups, until we have reached the desired number of clusters. This procedure is exactly what **agglomerative clustering** does, and it often mirrors what we are after in social sciences, where groups emerge, rather than spontaneously/randomly form.

agglomerative clustering

The question here is, of course: *how do we decide which two clusters to merge at each step*? This condition is called the **linkage criterion**. There are several conditions, but the most common is called **Ward clustering**, where we choose the pair of clusters that minimizes the variance. Other choices include minimizing the average distance between clusters (called **average linkage criterion**), or minimizing the maximum distance between clusters (called **complete linkage criterion**). In `sklearn`, the default

linkage criterion
Ward clustering
average linkage criterion
complete linkage criterion

setting is Ward linkage.

```
from sklearn.cluster import AgglomerativeClustering

agg = AgglomerativeClustering(n_clusters=10, n_jobs=-1)
clusters = agg.fit_predict(X)
```

CODE 29. Agglomerative clustering of 10 clusters with Ward linkage.

Agglomerative clustering does not give us cluster centroids, so if we need them, we have to compute them ourselves:

```
import numpy as np
centroids = np.array([X[clusters == c].mean(axis=0).A1 for c in range
    (k)])
```

CODE 30. Computing cluster centroids for agglomerative clustering.

We can further influence the clustering algorithm by providing information about how similar the data points are to each other before clustering. If we do that, clusters (i.e., data points) that are more similar to each other than others are merged earlier. Providing this information makes sense if our data points represent entities that are comparable to each other. For example, similar documents (i.e., about the same topics), people that share connections, or cities that are geographically close to each other. This information is encoded in a **connectivity matrix**. We can express the connectivity in binary form, i.e., either as 1 or 0. This hard criterion is telling the algorithm to either merge or never merge two data points. This notion is captured by an **adjacency matrix**: two countries either share a border, or they don't. If instead, we use continuous values in the connectivity matrix, we express the degree of similarity. Gradual similarity is the case, for example, in expressing the distance between cities.

connectivity matrix

adjacency matrix

In `sklearn`, we can supply the connectivity matrix as a `numpy` array when initializing the clustering algorithm, using `connectivity`.

```
agg = AgglomerativeClustering(n_clusters=10, n_jobs=-1, connectivity=
    C)
clusters = agg.fit_predict(X)
```

CODE 31. Agglomerative clustering of 10 clusters with Ward linkage and adjacency connectivity.

The only thing we need to ensure is that the connectivity matrix is square (i.e., it has as many rows as columns), and the dimensionality of both is equal to the number of documents, i.e., the connectivity matrix is a n-by-n matrix.

Figure 4 shows the effect of different clustering solutions on a data set from Hovy and Purschke (2018)

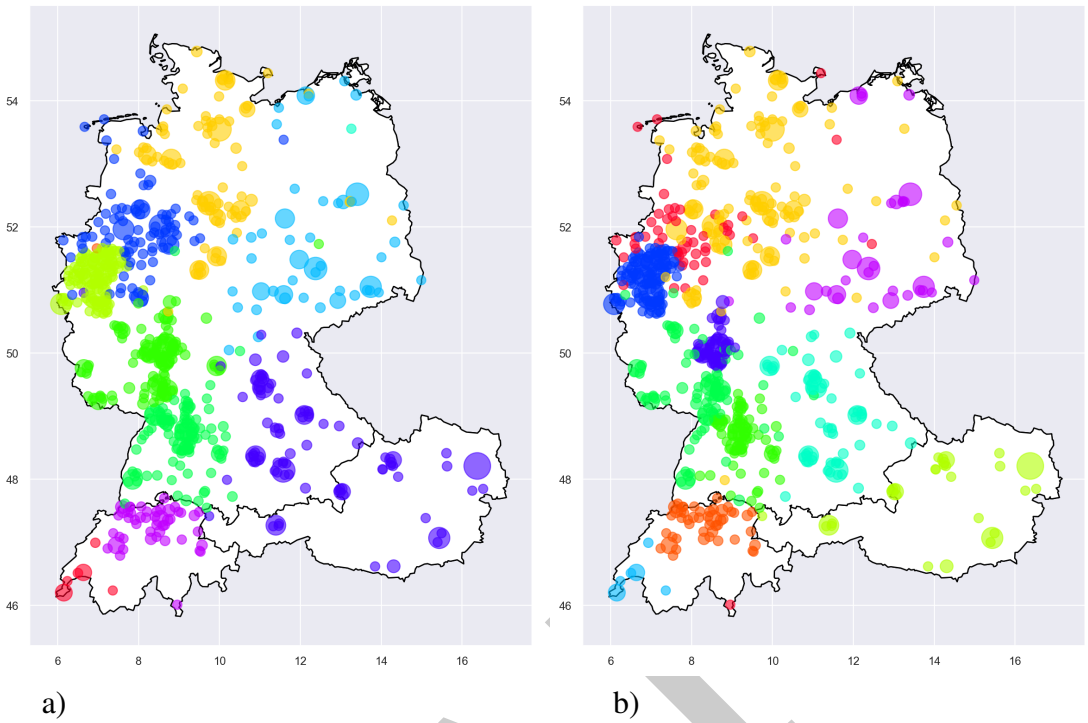


FIGURE 4. Example of clustering document representations of German-speaking cities with k-means (a) and agglomerative clustering with a distance matrix (b) into 11 clusters

2.3. Comparison. There are very clear trade-offs between the two clustering algorithms, as shows in Table 3.

	k-means	agglomerative
scalable	yes	no (up to about 20k)
repeatable result	no	yes
include external info	sort-of (initialization)	yes
good on dense clusters?	no	yes

TABLE 3. Comparison of k-means and agglomerative clustering.

Under most conditions, agglomerative clustering is the better choice. That is, until we have too many instances (say, more than 20,000). In those cases, it can become prohibitively slow, due to the many comparisons we have to make in each step. If we run into this problem, we might use k-means. However, we can initialize it with the cluster centroids derived via agglomerative clustering from a subset of the data. That way, we get a bit of the best of both worlds: stable, repeatable results, and fast convergence.

2.4. Choosing the Number of Clusters. So far, we have only compared which *method* is better for the data we have, but have quietly assumed that some outside factor determines the number of clusters. In many cases, though, the exact number of clusters is yet another parameter we need to choose.

To choose the optimal number of clusters, we can use various methods, each with their pros and cons. Here, we will focus on the **Silhouette method**. We fit different models with increasing numbers of clusters and measure the effect of the resulting solution on the Silhouette score. We select the number of clusters with the highest score. This score is composed of two parts. The first of them measures how compact each cluster is (i.e., the average distance of each instance in a cluster to all of its other members). The other score measures how well separated the individual clusters are (i.e., how far away each instance is on average from the nearest example in another cluster). The final metric is the average of this score across all cases in our data.

Silhouette method

In practice, we can implement this as follows:

```
from sklearn.metrics import silhouette_score

scores = []
for n_clusters in range(20):
    agg = AgglomerativeClustering(n_clusters=n_clusters, n_jobs=-1)
    clusters = agg.fit_predict(X)

    scores.append(silhouette_score(X, clusters))
```

CODE 32. Computing the Silhouette score for 20 different models.

We can then select the number of clusters corresponding to the highest score.

2.5. Evaluation. To test the performance of our clustering solutions, we can compare it to some known grouping (if available). In that case, we are interested in how well the clustering lines up with the actual groups. We can look at this overlap from two sides, and there are two metrics to do so: **homogeneity** and **completeness**.

homogeneity
completeness

Homogeneity measures whether a cluster contains only data points from a single gold standard group. Completeness measures how many data points of a gold standard group are in the same cluster. The difference is the focus. For the example of matching dialect regions, homogeneity is high if a cluster occurs only in one region. At the same time, completeness is high if the region contains only points from one cluster. Figure 5 shows an example of matching clusters to dialect regions.

We can also take a harmonic mean between the two metrics to report a single score. This metric is called the **V-score**. It corresponds to the precision/recall/F1 scores commonly used in classification tasks.

V-score

```
from sklearn.metrics import homogeneity_completeness_v_measure

h, c, v = homogeneity_completeness_v_measure(known, clusters)
```

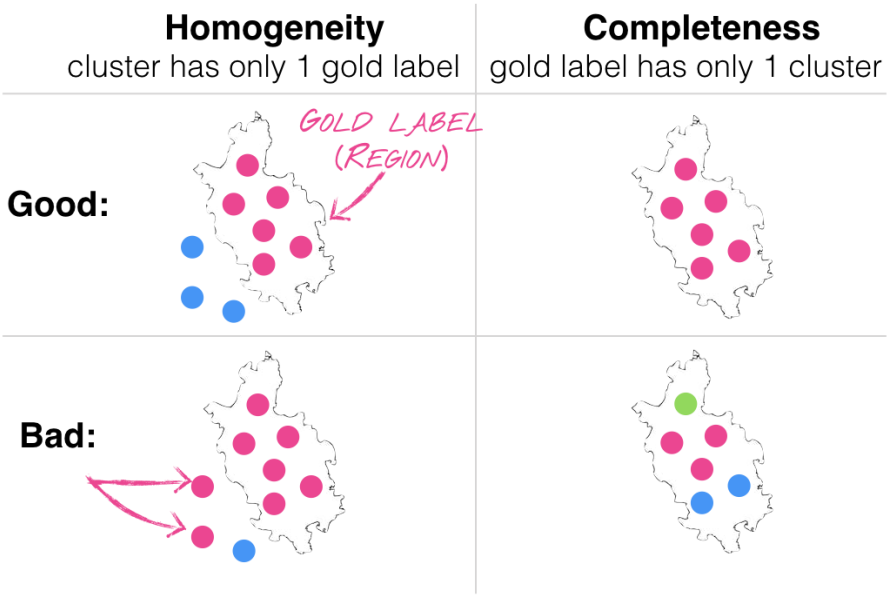


FIGURE 5. Homogeneity and completeness metrics for clustering solutions compared to a gold standard.

CODE 33. Computing homogeneity, completeness, and V-score for a clustering solution against the known grouping.

3. Language Models

Imagine a copywriter has produced several versions of a text advertisement. We want to investigate the theory that creative directors pick the most creative version of an ad for the campaign. We have the different draft versions, we have the final campaign, but how do we measure creativity? This is a big question, but if we just want a simple measure to compare texts, we could rank which of the texts are most unusual, compared to some standard reference. This is what **language models** do: for any text we give them, they return a probability of how likely/unusual this text is (with respect to some standard). In our example, we can use language models to assign each ad version a probability. A lower probability roughly corresponds to a more creative text (or at least one that is unlike most things we have seen). We can now check whether the texts with the lowest probabilities are indeed the ones that are picked most often by the creative director for the campaign, which would give us a way of measuring and quantifying "creativity".

language models

The interesting question is of course: "How do we compute the probability of a sentence?" There are indefinitely many possible sentences out there: you probably have produced several today that have never been uttered before in the history of ever. How would a model assign a probability to something infinitely large? We cannot just count and divide: after all, any number divided by infinity is going to be 0. In order to get around this problem, we divide a sentence into a sequence of smaller parts, compute their probabilities, and combine them. Formally, a language model is a probability function that takes any document S and a reference corpus θ and returns a value between 0 and 1, denoting how likely S is to be produced under the parameters θ.

Language models have a long history in NLP and speech processing (Jelinek and Mercer, 1980; Katz, 1987), because they are very useful in assessing the output quality of a system: it gives us a fast and easy way to rank the different possible outputs produced by a chat bot, or the best translation from a machine translation system, or a speech recognizer. Today, the best language models are based on neural networks (Bengio et al., 2003; Mikolov et al., 2010, inter alia), but they tend to require immense amounts of data. For most of our purposes here, a probability-based n-gram language model will work well.

More specifically, we will build a trigram LM based on **Markov chains**. Markov chains are a special kind of **weighted directed graphs** that describe how a system changes over time. Each state in the system is typically a point in discrete "time", so in the case of language, each state is a word. Markov chains have been used for weather prediction, market development, generational changes, speech recognition, NLP, and many other applications. They are also the basis of HMMs (Hidden Markov Models) and CRFs (Conditional Random Fields), which are widely used in speech recognition and NLP. Lately, they have been replaced in many of these applications by neural language models, which produce better output. However, probabilistic language models based on Markov chains are a simple and straightforward model to assess the likelihood

Markov chains

weighted directed graphs

of a document, and to generate short texts. In general, they assume that the probability of seeing a sequence (e.g., a sentence) can be decomposed into the product of a series of smaller probabilities:

$$P(w_1, w_2, ..., w_n) = \prod_i^N P(w_i|w_1, ...w_{i-1})$$

I.e., the probability of seeing a sentence is the probability of seeing each word in the sentence preceded by all others before it.

3.1. The Markov Assumption. In reality, this is a good idea. Each state or word depends on a long history of previous states: the weather yesterday and the day before influences the weather today, and the words you have read up to this one influence how you interpret the whole sentence. *Ideally*, we want to take the entire history up to the current state into account.

However, this quickly leads to exponentially complex models as we go on (just imagine the amount of words we would have to keep in memory for the last words in this sentence). The **Markov assumption** limits the history of each state to a fixed number of previous states. This limited history is called the **Markov order**. In a 0th order model, each state only depends on itself. In a first order model, each state depends on its direct predecessor. In a second order model, each state depends on its two previous states.

Markov assumption
Markov order

Of course this is a very simplifying assumption, and in practice, the higher the order, the better the model, but higher orders also produce sparser probability tables. And while there are some cases of long-range dependencies (for example when to open and close brackets around a very long sentence like this one), most of the important context is in the context words right before our target (at least in English).

3.2. Trigram LMs. In a trigram LM, we use a second order Markov chain. We first extract trigram counts from a reference corpus and then estimate trigram probabilities from them. In practice, it is enough to store the raw counts and compute the probabilities "on the fly".

A trigram (i.e., an n-gram with $n = 3$) is a sequence of 3 tokens, e.g., "eat more pie". Or, more general: $u\ v\ w$. A *trigram language model* (or trigram model) takes a trigram and predicts the last word in it, based on the first two (the history), i.e., $P(w|u, v)$. In general, the history or order of an n-gram model is $n - 1$ words. Other common n for language models are 5 or 7. The longer the history, the more accurate the probability estimates.

We will follow the notation in Mike Collins' lecture notes.[3] We are going to build a trigram language model that consists of a vocabulary $\mathcal{V}$ and a parameter $P(w|u, v)$ for each trigram $u\ v\ w$. We will define that the last token in a sentence X is STOP, i.e., x_n=STOP. We will also prepend $n - 1$ dummy token to each sentence, so we can count the first word, i.e., in a trigram model, x_0 and x_{-1} are set to $\star$.

[3] `http://www.cs.columbia.edu/~mcollins/lm-spring2013.pdf`

We first have to collect the necessary building blocks (the trigram counts) for the LM from a corpus of sentences. This corpus determines how good our LM is: bigger corpora give us better estimates for more trigrams. However, if we are only working within a limited domain (say, all ads from a certain agency), and want to rank sentences within that, the size of our corpus is less important, since we will have seen all the necessary trigrams we want to score.

Take the following sentence: `* * eat more pie STOP` The 3-grams we can extract are

```
[('*', '*', 'eat'),
 ('*', 'eat', 'more'),
 ('eat', 'more', 'pie'),
 ('more', 'pie', 'STOP')]
```

The $\star$ symbols allow us to estimate the probability of a word starting a sentence ($P(w|*,*)$), and the `STOP` symbol allows us to estimate what words end a sentence.

3.3. Maximum likelihood estimation (MLE). The next step is to get from the trigram counts to a probability estimate for a word given its history, i.e., $P(w|u,v)$. We will here use the most generic solution and use **maximum likelihood estimation**. We estimate the parameters of the statistical model that maximize the likelihood of the observed data (which in practice simply translates into: count and divide). We define:

maximum likelihood estimation

$$P(w|u,v) = \frac{c(u,v,w)}{c(u,v)}$$

where $c(u,v,w)$ is the count of the trigram and $c(u,v)$ the number of times the history bigram is seen in the corpus.

Instead of actually storing both trigram and bigram counts, we can make use of the fact that the bigrams are a subset of the trigrams. We can **marginalize out** the count of a bigram from all the trigrams it occurs in, by summing over all those occurrences, using the following formula:

marginalize out

$$P(w|u,v) = \frac{c(u,v,w)}{c(u,v)} = \frac{c(u,v,w)}{\sum_z c(u,v,z)}$$

E.g., in order to get the probability of $P(STOP|more,pie)$, we count *all* trigrams starting with "more pie".

3.4. Probability of a Sentence. Now that we have the necessary elements of a trigram language model (i.e., the trigram counts (u,v,w)) and a way to use these counts to compute the necessary probabilities $P(w|u,v)$, we can use the model to calculate the probability of an entire sentence based on the Markov assumption. The probability of a sentence is the product of all the trigram probabilities involved in it:

$$P(x_1,...,x_n) = \prod_{i=1}^{n} P(x_i|x_{i-2},x_{i-1})$$

I.e., we slide a trigram window over the sentence, compute the probability for each of them, and multiply them together. If we have a long sentence, we will get very small numbers, so it is better to use the logarithm of the probabilities again:

$$\log P(x_1, ..., x_n) = \sum_{i=1}^{n} \log P(x_i | x_{i-2}, x_{i-1})$$

Note that when using logarithms, products become sums, and division becomes subtraction, so the logarithm of the trigram probability can be in turn computed as

$$\log P(w|u, v) = \log c(u, v, w) - \log \sum_z c(u, v, z)$$

3.5. Smoothing. If we apply the LM only to the sentences in the corpus we used to collect the counts, we are done at this point. However, sometimes we want to compute the probability of a sentence that is *not* in our original corpus (for example, if we want to see how likely the sentences of another ad agency are).

Because language is so creative and any corpus, no matter how big, is limited, some perfectly acceptable three-word sequences will be missing from our model. This means that a large number of the possible n-grams will have a probability of 0 under our model, even though they should really get a non-zero probability. This is problematic, because any 0 count for a single trigram in our product calculation above would make the probability of the entire sentence 0 from thereon out, since any number multiplied by 0 is, well, 0:

$$P(x_1, ..., x_n) = P(x_1 | **) \cdot P(x_2 | * x_1) \cdot 0 \cdot ... = 0.0$$

smoothing
discounting
linear interpolation

Therefore, we need to modify the MLE probability computation to assign some probability mass to new, unseen n-grams. This is called **smoothing**. There are numerous approaches to smoothing, i.e., **discounting** (reducing the actual observed frequency) or **linear interpolation** (computing several n-grams and combining their weighted evidence). There is a whole paper on the various methods (Chen and Goodman, 1996), and it is still an area of active research.

Laplace-smoothing

The easiest smoothing, however, is **Laplace-smoothing**, also known as "add-1 smoothing". We simply assume that the count of any trigram, seen or unseen, is 1, plus whatever evidence we have in the corpus. It is the easiest and fastest smoothing to implement, and while it lacks some desirable theoretical properties, it deals efficiently with new words. So, for a trigram that contains an unseen word x, our probability estimate becomes:

$$P(w|u, x) = \frac{c(u, x, w) + 1}{c(u, x) + 1} = \frac{0 + 1}{\sum_z c(u, x, z) + 1}$$

Here, the resulting probability is 1, which seems a bit high for an unseen trigram. We can reduce that effect by choosing a smaller smoothing factor, say, 0.001, i.e., pretending we have seen each word only a fraction of the time. As a result, unseen

trigrams will have a much smaller probability.

Putting everything above together in code, we can estimate probabilities for any new sentence. Smoothing can be efficiently implemented by using the `defaultdict` data structure, which returns a predefined value for any unseen key. By storing the trigrams in a nested dictionary, we can easily sum over them to compute the bigram counts:

```
from collections import defaultdict
import numpy as np
import nltk

smoothing = 0.001
counts = defaultdict(lambda: defaultdict(lambda: smoothing))

for sentence in corpus:
    tokens = ['*', '*'] + sentence + ['STOP']
    for u, v, w in nltk.ngrams(tokens, 3):
        counts[(u, v)][w] += 1

def logP(u, v, w):
    return np.log(counts[(u, v)][w]) - np.log(sum(counts[(u, v)].
   values()))

def sentence_logP(S):
    tokens = ['*', '*'] + S + ['STOP']
    return sum([logP(u, v, w) for u, v, w in nltk.ngrams(tokens, 3)])
```

CODE 34. A simple trigram LM.

This code can be easily extended to deal with higher-order n-grams.

Coming back to our initial question about creativity, we could use a trained language model to assign probabilities to all the ad versions, and measure whether there is a correlation between unusual, low-probability versions and their eventual success.

3.6. Generation. Typically, Markov chain language models are used to predict future states, but we can also use it to **generate text** (language models and Markov Chains are generative graphical models). In fact, Andrey Markov first used Markov Chains to predict the next letter in a novel. We will do something similar and use a Markov chain process to predict the next word, i.e., to generate sentences. **generate text**

This technique has become a bit infamous, since it was used by some people to generate real-looking, but meaningless scientific papers (that actually got accepted!), and is still used by spammers in order to generate their emails. However, there are also much more innocent use cases, such as generating real-looking nonce stimuli for

an experiment, or for a light-hearted illustration of the difference between two political candidates, by letting their respective LMs "finish" the same sentence.

We can use the counts from our language model to compute a probability distribution over the possible next words, and then sample from that distribution according to the probability (i.e., if we sampled long enough from the distribution of a word $P(w|u,v)$, we would get the true distribution over the trigrams).

```
import numpy as np

def sample_next_word(u, v):
    keys, values = zip(*counts[(u, v)].items())
    values = np.array(values)
    values /= values.sum()
    return keys[np.argmax(np.random.multinomial(1, values))]

def generate():
    result = ['*', '*']
    next_word = sample_next_word(result[-2], result[-1])
    result.append(next_word)
    while next_word != 'STOP':
        next_word = sample_next_word(result[-2], result[-1])
        result.append(next_word)

    return ' '.join(result[2:-1])
```

CODE 35. A simple trigram sentence generator.

This generates simple sentences. A trigram model is not very powerful, and in order to generate real-sounding sentences, we typically want a higher order, which then also means a much larger corpus to fit the model on.

4. Topic Models

Imagine we have a corpus of restaurant reviews and want to learn what people are mostly talking about: what are their main complaints (price, quality, hygiene?), what types of restaurants (upscale, street food) are they discussing, and what cuisines (Italian, Asian, French, etc.) do they prefer? TF-IDF terms can give us individual words and phrases that hint at these things, but they do not group together into larger constructs. What we want is a way to automatically find these larger trends.

Topic models

Topic models are one of the most successful and best-known applications of NLP in social sciences (not in small part due to its availability as a package in `R`). Their advantage is that they allow us to get a good high-level overview of the things that are being discussed in our corpus. What are the main threads, issues, and concerns in the texts?

What we ultimately want is a distribution over topics for each document. E.g., our first document is 80% topic 1, 12% topic 2, and 8% topic 3. In order to interpret these topics, we want for each topic a distribution over words. E.g., the five words most associated with topic 1 are "pasta", "red_wine", "pannacotta", "aglio", and "ragù". Based on these words, we can give it a descriptive label that sums up the words (here, for example "Italian cuisine"). This also gives us the basic elements: words, documents, and topics.

The most straightforward way to express the intuition above mathematically is through probabilities. The three elements can be related to each other in two conditional probability tables: one that tells us how likely each word is for each topic ($P(\text{word}|\text{topic})$), and one that tells us how likely each topic is for a specific document ($P(\text{topic}|\text{document})$). In practice, the distribution of words over topics is usually just called z, and the distribution over topics for each document is called θ. We also have a separate distribution θ_i for every document, rather than one that is shared for the entire corpus. After all, we do not expect that a document about a fast-food joint hits upon the same issues as a review of the swankiest 3-star Michelin place in town. Figure 6 shows the two quantities as graphical distributions.

generative probabilistic models
Latent Dirichlet Allocation (LDA)

The term "topic model" covers a whole class of **generative probabilistic models**, with plenty of variations, but the most well-known is **Latent Dirichlet Allocation (LDA)** (Blei et al., 2003). So in the following, we will use the two terms interchangeably.

The operative word here is "generative": probabilistic models can be used to describe data, but they can just as well be used to *generate* data, as we have seen with language models. Topic models are very similar. In essence, a topic model is a way to imagine how a collection of documents was generated word by word. They have a Markov horizon of 0 (i.e., each word is independent from all the ones before), but instead each word is conditioned on a topic. In other words: with a topic model, rather than conditioning on the previous words (i.e., the history), we first choose a topic and

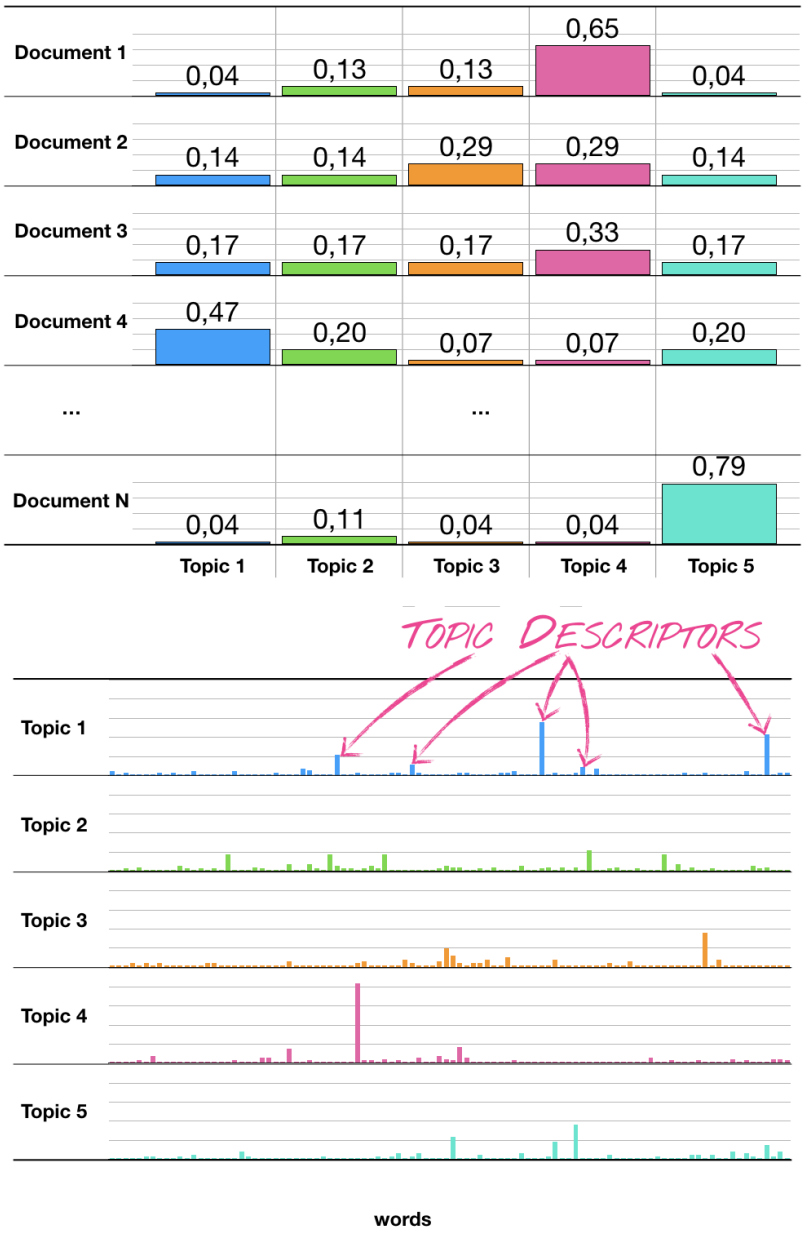


FIGURE 6. Graphical representation of the topic model parameters $\theta = P(\text{topic}|\text{document})$ (top) and $z = P(\text{word}|\text{topic})$ (bottom). Topic descriptors are the top k words with the highest probability for each topic in z

then select a word based on the topic. How this is supposed to have happened is described in the **generative process**. This is why LDA is a "generative model". The steps

generative process

in this generative story for how a document came to be are fairly simple: for each word in a document, we first choose the topic of that slot, and then look up which word from the topic should fill the slot. Or, more precisely, for a document d_i:

(1) Set the number of words N by drawing from a **Poisson distribution** **Poisson distribution**
(2) Draw a multinomial topic distribution θ over k topics from a **Dirichlet distribution** **Dirichlet distribution**
(3) For $j \in N$:
    - Pick a topic t from the multinomial distribution θ
    - Choose word w_j based on its probability to occur in the topic according to the conditional probability $z_i j = P(w_j|t)$

No, this is of course not how documents are generated in real life, but it is a useful abstraction for our purposes. Because if we assume that this *is* indeed how the document was generated, then we can reverse-engineer the whole process to find the parameters that did it. And those are the ones we want. The generative story is often depicted in **plate notation** (Figure 7) We have already seen θ and z. The first describes the **plate notation**

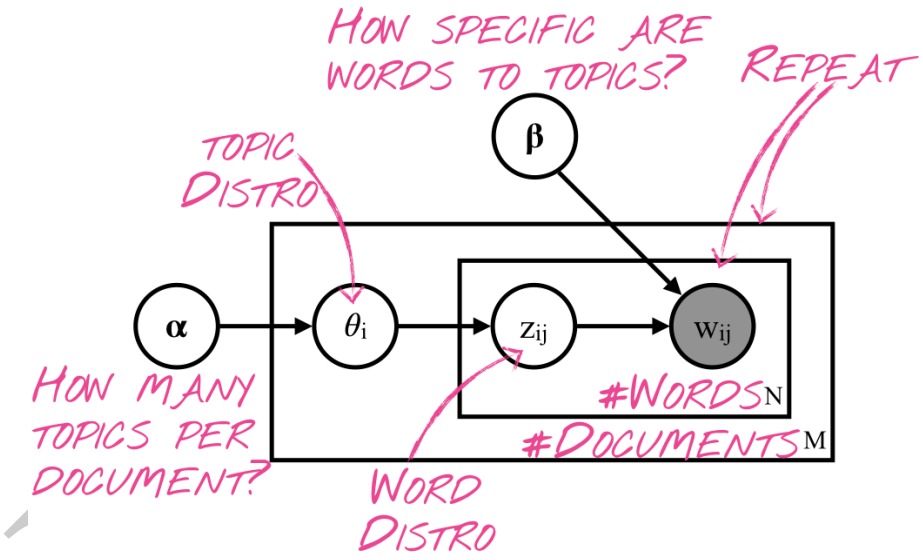


FIGURE 7. Plate notation of LDA model.

probability of selecting a particular topic when sampling from a document. The second describes the chance of selecting a particular word when sampling from a given topic. Each of those probability distributions have a **hyperparameter** attached to them, α and β, respectively. α is the **topic prior**: it is the parameter for the Dirichlet process (a process that generates probability distributions) and controls how many topics we **hyperparameter** **topic prior**

expect each document to have on average. If we expect every document to have several or all of the topics present, we choose an α value at 1.0 or above, leading to a rather uniform distribution. If we know or suspect that there are only a few topics per document (for example because the documents are quite short), we use a value smaller than 1.0, leading to a very peaked distribution. β is the **word prior**: it controls how topic-specific we expect the words to be. If we think that words are very specific to a particular topic, use a small β value (0.01 or less), if we think they are general, we use values up to 1.0 We can imagine both of these parameters a bit like entropy: the lower the value, the more peaked the underlying distribution.

word prior

Now, we do not have the two tables yet, so we need to infer them from the data. This is usually done using either **Gibbs sampling** or **Expectation Maximization** (Dempster et al., 1977). We do not have the space here to go in depth of how these algorithms work (gentle introductions to them are Resnik and Hardisty (2010) and Hovy (2010), respectively). Both are essentially guaranteed to find better and better parameter settings the longer we run them (i.e., the models become more and more likely to have produced the data we see). The question is of course: where do we start? In the first round, we essentially set all parameters to random values, and see what happens. Then we update the parameters in order to improve the fit with the data and run again. We repeat this until we have reached a certain number of iterations, or the parameters stop changing much from iteration to iteration, i.e., the model fits the data more and more. We can measure the fit of the model to the data (also called the **data likelihood**) by multiplying together all the probabilities involved in generating the corpus.

Gibbs sampling

Expectation Maximization

data likelihood

4.1. Caveats. The results depend strongly on the initialization and the parameter settings. There are a couple of best practices and rules of thumb (see also Boyd-Graber et al. (2014)):

(1) thoroughly preprocess your data: lowercase the text, use lemmatization, remove stopwords, replace numbers and user names, and join collocations (and document your choices).
(2) use a minimum document frequency of words: exclude all words that are rare (a word that we have seen 5 times in the data could just be a name or a common misspelling). Depending on your corpus size, a document frequency of 10, 20, 50, or even 100 is a good starting point.
(3) maximum document frequency of words: a word that occurs in more than 50% of all documents has very little discriminative power, so you might as well remove it. However, in order to get coherent topics, it can be good to go as low as 10%
(4) choose a good number of iterations: when is the model done fitting? While the model will always improve the data likelihood with more iterations, we do not want to wait indefinitely. Choosing a number that is too small, however, will result in an imperfect fit. After you are happy with the other parameter choices, extend the number of iterations and see whether the results improve.

In either case, you will have to look at your topics and decide whether they make sense. This can be non-trivial: a topic might perfectly capture a topic you are not aware of (e.g., the words "BLEU, Bert, encoder, decoder, transformer" might look like random gibberish, but they perfectly encapsulate papers on machine translation).[4] Because topics are so variable from run to run, they are not stable indicators for dependent variables: we cannot use the output of a topic model to predict something like ratings.

4.2. Implementation. In order to implement LDA in Python, there are two possibilities, `gensim` and `sklearn`. We will look at the `gensim` implementation here, and run it on a sample of wine descriptions to find out what sommeliers talk about.

```
from gensim.models import LdaMulticore, TfidfModel
from gensim.corpora import Dictionary
import multiprocessing
```

CODE 36. Code for loading topic models from `gensim`.

```
dictionary = Dictionary(instances)
dictionary.filter_extremes(no_below=100, no_above=0.1)

ldacorpus = [dictionary.doc2bow(text) for text in instances]

tfidfmodel = TfidfModel(ldacorpus)

model_corpus = tfidfmodel[ldacorpus]
```

CODE 37. Extracting and limiting the vocabulary, and transforming the data into TF-IDF representations of the vocabulary.

```
num_passes = 10
num_topics = 20
# find chunksize to make about 200 updates
chunk_size = len(model_corpus) * num_passes/200

model = LdaMulticore(model_corpus,
                     id2word=dictionary,
                     num_topics=num_topics,
                     workers=min(10, multiprocessing.cpu_count()-1),
                     passes=num_passes,
                     chunksize=chunk_size
                    )
```

CODE 38. Training the LDA model on the corpus on multiple cores.

[4] Thanks to Hanh Nguyen for the example.

This will give us a model we can then use on the corpus in order to get the topic proportions for each document:

```
topic_corpus = model[model_corpus]
```

CODE 39. Transforming the corpus into topic proportions.

We can also examine the learned topics:

```
model.print_topics()
```

CODE 40. Showing the topic words.

Unfortunately, these are formatted in a way that is a bit hard to read:

```
[(0,
  '0.011*"rose" + 0.010*"fruity" + 0.010*"light" + 0.009*"orange" +
   0.009*"pink" + 0.008*"lively" + 0.008*"balance" + 0.008*"mineral"
   + 0.008*"peach" + 0.007*"color"'),
  ...
]
```

We can use regular expressions to pretty this up a bit:

```
import re
topic_sep = re.compile(r"0\.[0-9]{3}\*")

model_topics = [(topic_no, re.sub(topic_sep, '', model_topic).split('
    + ')) for topic_no, model_topic in
                model.print_topics(num_topics=num_topics, num_words
   =5)]

descriptors = []
for i, m in model_topics:
    print(i+1, ", ".join(m[:5]))
    descriptors.append(", ".join(m[:2]).replace('"', ''))
```

CODE 41. Enumerate the topics and print out the top 5 words for each.

This gives us a much more readable format:

```
1 "rose", "fruity", "light", "orange", "pink"
2 "pretty", "light", "seem", "mix", "grow"
3 "tobacco", "close", "leather", "dark", "earthy"
4 "lime", "lemon", "green", "zesty", "herbal"
5 "pepper", "licorice", "clove", "herb", "bright"
6 "toast", "vanilla", "grill", "meat", "jam"
7 "jammy", "raspberry", "chocolate", "heavy", "cola"
8 "age", "structure", "year", "wood", "firm"
```

```
9 9 "pear", "apple", "crisp", "white", "attractive"
```

4.3. Selection and Evaluation. The first question with topic models is of course: *How many topics should I choose?* This is a good question, and in general, more likely models should also have more coherent and sensible topics, but that is a rough and tenuous equivalence. Which brings us to the question of **topic coherence**. Ideally, we want topics that are "self-explanatory", i.e., topics that are easy to interpret by anyone who sees them. A straightforward way to test this is through an **intrusion test**: take the top 5 words of a topic, replace one with a random word, and show this version to some people. If they can pick out the random intruder word, then the other words must have been coherent enough as a topic to make the intruder stick out. By asking several people and tracking how many guessed correctly, we can compute a score for each topic. Alternatively, we can use easy-to-compute measures that correlate with coherence for all pairs of the topic descriptor words, to see how likely we are to see these words together, as opposed to randomly distributed.

topic coherence

intrusion test

There are many measures we can use (Stevens et al., 2012), but here, we will focus on two: The UMass evaluation score (Mimno et al., 2011), which uses the log probability of word co-occurrences; and the C_v score (Röder et al., 2015), which uses the normalized pointwise mutual information and cosine similarity of the topic words. We can use these measures to compute a coherence score for each number of topics on a subset of the data, and then choose the number of topics that gave us the best score on both (or on the one we prefer).

In Python, we can choose these with the `CoherenceModel`

```
1 from gensim.models import CoherenceModel
2
3 coherence_values = []
4 model_list = []
5 for num_topics in range(5, 21):
6     print(num_topics)
7     model = LdaMulticore(corpus=sample, id2word=dictionary,
    num_topics=num_topics)
8     model_list.append(model)
9     coherencemodel_umass = CoherenceModel(model=model, texts=
    test_sample, dictionary=dictionary, coherence='u_mass')
10
11    coherencemodel_cv = CoherenceModel(model=model, texts=test_sample
    , dictionary=dictionary, coherence='c_v')
12
13    coherence_values.append((num_topics, coherencemodel_umass.
    get_coherence(), coherencemodel_cv.get_coherence()))
```

CODE 42. Choosing topic number via coherence scores.

That will give us a relatively good idea of the best number of topics, as we can see in Figure 8.

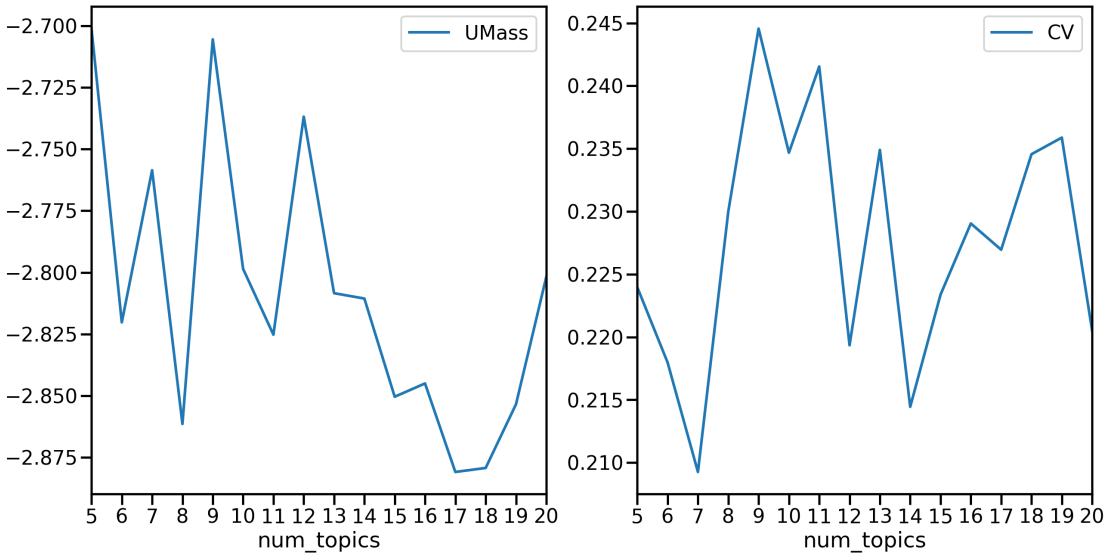


FIGURE 8. Coherence scores for different numbers of topics

Even after we settle on a number of topics, there is still quite a lot of variation. If we run 5 topic models with the same number of topics on the same data, we will get 5 slightly different topic models, because we initialize the models randomly. Some topics will be relatively stable between these models, but others might vary greatly. So how do we know which model to pick? The data likelihood of a model can also be used to calculate the **entropy** of the model, which in turn can be used to compute the **perplexity**. We can compare these numbers and select the model with the best perplexity. Entropy and perplexity are model-inherent measures, i.e., they only depend on the parameters of the model. That means we can use a grid search over all the possible combination of parameter values to find the best model.

entropy

perplexity

4.4. Adding Structure. A popular variant of LDA is the **structural topic model** (STM) by Roberts et al. (2013). This model has become very popular in computational social science, since we often have access to external factors (covariates) associated with the text, such as the author of the text, when it was written, or which party published it. More often than not we are interested in the effects of these factors on our topics. A simple way is to aggregate the topic distributions by each of these factors. However, a more principled approach is to algorithmically model the influence of the external factors on the topic distributions.

structural topic model

The original STM is again only available as an `R` package, but `gensim` added a version of the structural topic model as **Author Topic Model**. The interface is essentially the same as for the regular LDA model, with one addition: we need to provide a

Author Topic Model

dictionary that maps from each value of our external factor to the documents associated with it. In the original context, this is a mapping from scientific authors to the papers they were involved in writing. In our example, let's use the country of origin for each wine as covariate: that way, we can learn what the dominant flavor profiles are for each country (see Figure 9). As we can see, each category receives a dominant topic.

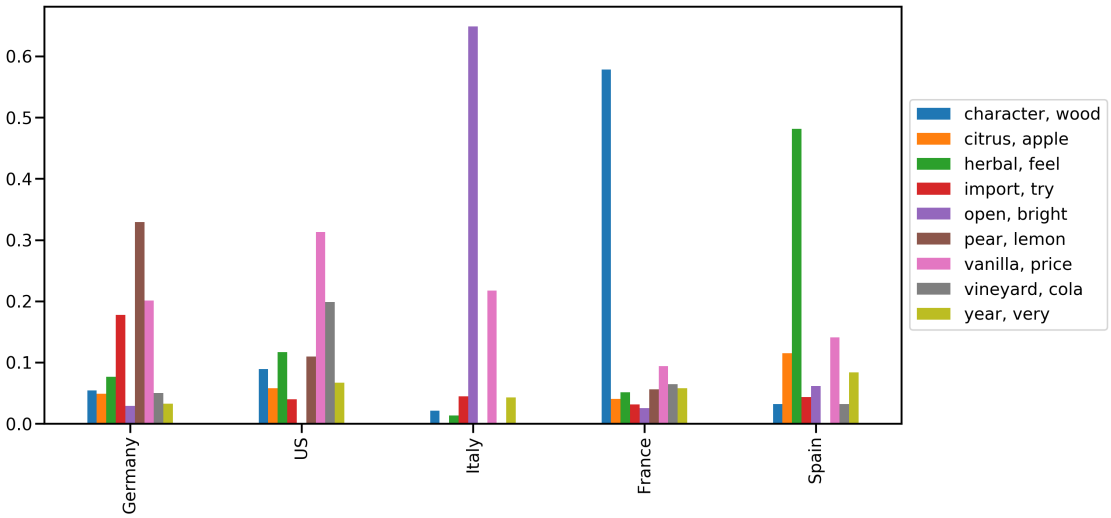


FIGURE 9. Topic distribution by country

The implementation in Python is very similar to the regular LDA model. The only thing we need to do before we can start training, is to map each covariate (here, a country) to the index of all documents it is associated with. In our case, each document has only one covariate, but we can easily associate each covariate with a number of documents (for example if we have scientific papers written by several authors).

```
from collections import defaultdict
author2doc = defaultdict(list)

for i, country in enumerate(df.country):
    author2doc[country].append(i)

```

CODE 43. Creating the mapping from covariates to document indices.

Once we have this mapping, we can run the model.

```
from gensim.models import AuthorTopicModel
from gensim.test.utils import temporary_file

with temporary_file("serialized") as s_path:
    author_model = AuthorTopicModel(
```

```
        model_corpus,
        author2doc=author2doc,
        id2word=dictionary,
        num_topics=9,
        serialized=True,
        serialization_path=s_path,
        alpha=0.5
    )

    author_model.update(model_corpus, author2doc)
```

CODE 44. Running the Author LDA model.

The results we get are similar to the regular LDA model (again, we use regular expressions to clean things up a bit):

```
1 "raspberry", "not", "make", "vanilla", "chocolate"
2 "tone", "bright", "back", "velvety", "smooth"
3 "open", "feel", "tobacco", "deliver", "close"
4 "character", "age", "structure", "wood", "fruity"
5 "green", "apple", "zesty", "herbal", "body"
6 "peach", "honey", "lemon", "apple", "pear"
7 "flesh", "bouquet", "mouth", "mingle", "rind"
8 "pair", "mouthfeel", "white", "easy", "exotic"
9 "cheerful", "guava", "mouthful", "underripe", "blast"
```

As we have seen, if we visualize these topic proportions per country (see Figure 9), the countries each get a different profile, corresponding to their dominant styles of wine.

4.5. Adding Constraints. Even when setting the hyperparameters well and having enough data to fit, there is a good chance that topics which we know should be separate are merged by the model. One way to keep them apart is to tinker with z, by adding priors to some of the words. This will nudge the model towards some distributions and away from others. In practice, we are preventing the model from exploring some parameter configurations that we know will not lead to a solution we want. Jagarlamudi et al. (2012) introduced a straightforward implementation of such a **guided LDA**.

guided LDA

Blodgett et al. (2016) used a fixed prior distribution over topics based on the known distribution of different ethnicities in the cities they investigated. By using the same number of topics as there were ethnicities in the data, they were able to capture ethnolects and regional terms.

4.6. Topics vs. Clusters. A relatively frequent question at this point is "Why should I use a topic model if I could just cluster word embeddings?" The two approaches do indeed produce similar results: a separation of the data into larger semantic groups. The differences are somewhat subtle: word embeddings capture semantic similarity within the defined window size, topic models take the entire document into

account. If the documents are short (e.g., Tweets, reviews) and the window size of the embedding model is sufficiently wide, the two are equivalent. However, with longer document types, word embeddings capture only general semantic fields (i.e., what are all the things people talk about in the data), not any specific to a particular document (i.e., what is the distribution of things people talk about in each document). On the other hand, clustering embeddings is relatively stable and will produce the same results every time, whereas the output of topic models varies with every run.

Lately, there has been an increased interest in neural (rather than probabilistic) topic models. They seem to be more flexible, result in more coherent topics, and are applicable to multiple languages (Das et al., 2015; Srivastava and Sutton, 2017; Dieng et al., 2019; Bianchi et al., 2020). However, there are no widely available Python libraries — yet.

5. Retrofitting

Let's say we have a corpus of texts from various political parties, and we learn word embeddings. In addition to the semantic similarities, these embeddings reflect some of the biases and attitudes reflected in the texts. However, they do not necessarily reflect the party divisions. We can include this information by using document embeddings for the various parties. However, this requires us to have the information at the time of training, and adds a layer of complexity to the training process (i.e., we might need more training data than for pure word embeddings). Instead, we might want to manipulate the existing word embeddings, keeping the original semantic distinctions, but adding further, non-linguistic information we have. In the same vein, if we have learned some document embeddings of firms based on their online self-descriptions, we might want to incorporate external knowledge we have about the markets they are in, or the types of companies they are (privately or publicly owned, etc.)

So far, we have looked at embeddings as a way to represent the *linguistic* similarity between words and documents in terms of spatial distance. However, especially when our documents represent entities in the real world, we might have additional external, non-linguistic knowledge about the entities that we might want to incorporate. In order to do this, we use an algorithm called **retrofitting** (Faruqui et al., 2015; Hovy and Fornaciari, 2018).

retrofitting

Retrofitting was originally introduced as a way to refine existing word embedding vectors to let them reflect semantic similarities that were not apparent in the training data, but instead came from external lexicons Faruqui et al. (2015). These lexicons included external ontologies, such as WordNet Miller (1995) or the Paraphrase Data Base PPDB Ganitkevitch et al. (2013) to bring synonyms (e.g., "flat" and "apartment") even closer together in embedding space.

Formally, we create a set Ω that contains tuples of words that belong to the same equivalence group, for example $\Omega = \{(apartment, flat)\}$. During retrofitting, we iteratively update the word embeddings of words within the same class (as defined in Ω) to increase the cosine similarity between them. This creates a retrofitted matrix $\hat{D}_{train}$ of the original word embedding matrix D_{train}. The update for a word embedding d_i is simply a weighted combination of two things: the original word embedding, and the average vector over all the embeddings of its neighbors:

$$\hat{d}_i = \alpha d_i + \beta \frac{\sum_{j:(i,j)\in\Omega} \hat{d}_j}{N}$$

where d_i is the original embeddings vector, $N = |\{\forall j : (i,j) \in \Omega\}|$ is the set of all embeddings in the same equivalence group, and α and β are hyper-parameters that control the trade-off between the original embeddings space and the updates from the neighboring embeddings during retrofitting. In Faruqui et al. (2015), they set $\alpha = \beta$. In contrast, we can also define

$$\beta = 1 - \alpha$$

By varying α from 0 to 1, we can control the strength of the retrofitting process, effectively trading off the influence of the original embeddings versus the vectors of the equivalence classes. So $\alpha = 1$ simply reproduces the original embeddings matrix, i.e., $\hat{D} = D$, whereas $\alpha = 0$ only relies on the retrofitting updates from the neighborhood after the initialization.

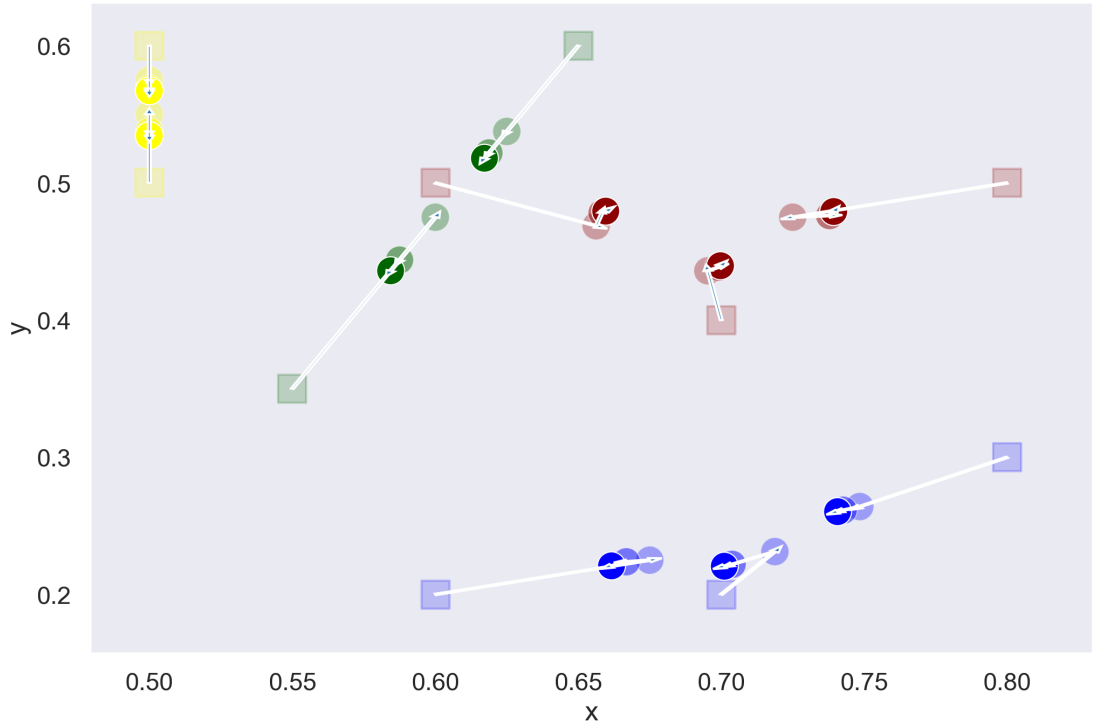


FIGURE 10. Example of retrofitting for 10 words in embeddings space, equivalence class represented by color.

With each iteration of the retrofitting, we move the target word according to the current position of its neighbors. Since those neighbors get moved as well, the average in each round will be slightly different, but converge over time. Because we always use the original vector as part of the update, the retrofitting does not end up collapsing all embeddings into the same point. Figure 10 shows the effect of retrofitting on 10 word embeddings, with equivalence classes colored by class. The squares represent the starting point, and the circles show the position at each iteration of the retrofitting. Words belonging to the same equivalence group get drawn closer to each other in embeddings space.

If we are adjusting document embeddings, and our equivalence lexicon Ω is based upon the membership of each document in an outcome class (say, all positive vs. all negative reviews) such that all members of one class are connected, we essentially make

the two classes more distinct, which makes them easier to distinguish. We will come back to this in the chapter on classification (see chapter 11).

The code for retrofitting is extremely simple: it only requires a matrix of N embeddings, and an N-by-N adjacency matrix of neighbors (which is programmatically faster and easier to handle than a dictionary):

```
from copy import deepcopy
import numpy as np
import sys

def retrofit(vectors, neighbors, normalize=False, num_iters=10, alpha=0.5):
    beta = 1-alpha
    N, D = vectors.shape

    new_vectors = deepcopy(vectors)

    if normalize:
        for c in range(N):
            vectors[c] /= np.sqrt((vectors[c]**2).sum() + 1e-6)

    # run retrofitting
    print("retrofitting", file=sys.stderr)

    for it in range(num_iters):
        print("\t{}:".format(it + 1), end=' ', file=sys.stderr)
        # loop through every instance
        for c in range(N):
            print(".", end='', file=sys.stderr)
            instance_neighbours = neighbors.get(c, [])
            num_neighbours = len(instance_neighbours)
            #no neighbours, pass - use data estimate
            if num_neighbours == 0:
                continue

            new_vectors[c] = alpha * vectors[c] + beta * new_vectors[instance_neighbours].sum(axis=0)

        print('', file=sys.stderr)

    return new_vectors
```

CODE 45. A simple trigram LM.

Applications of retrofitting can also include geographic information, such as the adjacency of cells, see Hovy et al. (2019); Purschke and Hovy (2019).

Part III

Prediction
Using patterns in the data

In the previous part, we looked at ways of exploring textual data, to find out what kind of structures are in there. We will build on some of that knowledge in this part, but we will use it to infer information for new, unseen data using **prediction**.

prediction

Prediction is useful for inferring both linguistic structure (parts of speech, syntax, NER, discourse structure), as well as a multitude of social constructs, signaled in language. Sentiment analysis is probably the most well-known application. However, there is virtually no limit in the kind of things we can predict from text: power, trust, misogyny, age, gender, etc.

Prediction is a core part of machine learning, and essentially involves showing the computer lots of examples of inputs (i.e., in our case documents) and the correct output label for them (for example, *spam* or *no spam*, or *positive*, *negative*, *neutral*). Note that we can have any number of possible output labels (**classes**).

classes

Because we try to predict the class labels of new input, prediction is synonymous with **classification**. If instead of classes, we predict a continuous numerical value, we refer to it as **regression**. In social sciences, there are some cases where we would like to do regression, but there are practically no use cases in NLP. The principles described below are largely analogous for regression, but we will not cover it in detail.

classification
regression

Prediction is a handy application of machine learning, both for NLP and social sciences. It allows us to infer variables of interest (power relations, buying intent, etc.) and complete missing variables (e.g., age, gender), based on language. The principles behind these models are relatively simple. With sufficient data and ever more powerful models, it is easy to get carried away in the possibilities.

We will look at the basic principles behind classification, how to set up experiments in a realistic way (see page 113 and 115) to evaluate the performance (see page 117), test the significance of our performance results (see page 120), and prevent overfitting (see page 124). We will then look at how to improve the predictive performance of the models with a variety of techniques (see page 133).

In the last few sections of the books, we will take a look at neural networks. This is a booming research area (even though the basic idea is already from the 1950s), so we will only be able to cover the basics (see page 160), as well as a number of model architectures that have shown themselves especially useful for text analysis (see pages 167, 173, 176).

However, since language is one of the most individual human capabilities, it also reveals a lot about the person using it. The predictive models we build can expose people's profiles and characteristics without them being aware of it. With great power, there comes great responsibility, so before we delve deeper into the algorithmic side of prediction, let's look at the ethical aspects of it.

1. Ethics, Fairness, and Bias

You have a data set with survey answers. For some participants, you have information about their age and gender. However, you ran several trials, and in some of them, you forgot to have subjects provide demographic information, so now you have missing data. It then turns out that controlling for gender would be a handy thing to do for your study. You decide to train a classifier on the subset of the data where gender information is available, and then apply the classifier to the rest of the data. That way, you can impute the missing demographic information. Your classifier performs well, and you complete the study, now able to control for the effect of gender.

As you look at the data again, though, you realize that it has a bias. Most of the participants who did provide gender information were women, so the classifier is much better at identifying women. It also suggests women more often. You now suspect that the gender distribution on the entire data set is skewed towards predicted female participants. You also realize that you have essentially created a tool that can infer gender where people did not provide it. Maybe they just forgot, but maybe they did not want to reveal this information. Is it ok to still use your tool on the data? Obviously, you are only interested in answering a scientific question, but you realize that someone could use your tool with much less honorable motives.

With great (predictive) power comes great responsibility, and especially when working with language, there are a number of ethical questions that arise. There are no hard and fast rules for everything, and the topic is still evolving, but several topics have emerged so far. While the overview here is necessarily incomplete, it is a starting point on the issue.[1]

Originally, machine learning and NLP were about solving fairly artificial problems on small data sets, with the promise of doing it on larger data at some later point. And while there has always been a certain amount of skepticism and worry about AI's power, these worries were largely theoretical. In essence, there was not enough data and computational power for these systems to make an impact on people's life.

With the recent availability of large amounts of data, and the universal application of ML, this point has finally arrived. With the focus on making useful tools, we have moved away from explanatory and descriptive models (to understand *why* they returned a certain result), towards predictive models. They are harder to analyze, but produce excellent predictions.

It now turns out that one of the reasons why models have gotten so good at prediction is because they pick up on things that they were not meant to exploit. Embedding models reflect ethnic and gender stereotypes (Bolukbasi et al., 2016), bail decision predictions are majorly influenced by the defendants' ethnicity (Angwin et al., 2016), automatic captioning and smart speakers do not work for people with non-standard

[1]This chapter is based on the work by Hovy and Spruit (2016), as well as the ACL workshops on Ethics in NLP.

language varieties (Tatman, 2017; Harwell, 2018), and skin cancer detectors work only on white skin (Adamson and Smith, 2018).

bias

All of these unintended consequences are examples of **bias**: a systematic difference from the truth. These biases can arise from the data, the models, or the research design itself.

Bias is not necessarily a problem a priori. In the behavioral psychology tradition, biases are mental shortcuts that help us save time and energy. In Bayesian statistics, the prior is a bias: it captures what we expect to encounter before we have any evidence. Expecting someone from Germany to have a German accent in English is a bias, but it means we won't be surprised if they do have an accent, and understand them better. A bias becomes problematic, though, when we trust it more than the data, or when we let it govern decisions.

In language data, demographic biases are powerful, since we use language to consciously and subconsciously signal who we are. And so language reflects a lot of information about our age, gender, ethnicity, region, personality, and even things like our profession and income bracket.

privacy

This leads to a second problem with better and better predictive models, namely **privacy**: at this point, we can use NLP systems that exploit the signals in our language use to predict all of the above features: people's age (Rosenthal and McKeown, 2011; Nguyen et al., 2011), gender (Alowibdi et al., 2013; Ciot et al., 2013; Liu and Ruths, 2013), personality (Park et al., 2015), job title (Preoţiuc-Pietro et al., 2015a), income (Preoţiuc-Pietro et al., 2015b), and much more (Volkova et al., 2014, 2015).

dual use

Being able to predict arbitrary demographic and socio-cultural attributes puts practitioners in a moral quandary. On the one hand, we want the best tools to answer our questions and make generalizable claims about the world. On the other hand, we do not want to develop tools that could be misappropriated for nefarious goals. This potential for abuse is the third problem, called **dual use**.

1.1. Sources of Bias.

1.1.1. *Data.* There are two ways in which data can introduce bias into our work: data selection and annotation. The selection of a demographically not-representative data set introduces **selection bias**. Annotation decisions made by the coders of our data introduce **annotation bias**.

selection bias
annotation bias

When choosing a text data set to work with, we are also making decisions about the demographic groups represented in the data. As a result of the demographic signal present in language, any data set carries a **demographic bias**, i.e., latent information about the demographic groups present in it. As humans, we would not be surprised if someone who grew up hearing only their dialect would have trouble understanding other people. So if our data set is dominated by the "dialect" of a specific demographic group, we should not be surprised that our models have problems understanding others.

demographic bias

Most data sets have some kind of built-in bias, and in many cases, it is benign. It becomes problematic when these bias negatively affect certain groups, or advantage others. On a biased data sets, statistical models will overfit to the presence of specific linguistic signals that are particular to the dominant group. As a result, the model will work less well for other groups, i.e., it leads to **exclusion** of demographic groups. **exclusion**

Concretely, Hovy and Søgaard (2015) and Jørgensen et al. (2015) have recently shown the consequences of exclusion for NLP. POS models have a significantly lower accuracy for young people, as well as ethnic minorities, vis-à-vis the dominant demographics in the training data. Apart from exclusion, these models will pose a problem for future research. Given that a large part of the world's population is currently under 30,[2] such models will degrade even more over time, and ultimately not meet the needs of its users.

This issue also has severe ramifications for the general applicability of any findings using these tools. In psychology, most studies are based on college students, a very specific demographic: western, educated, industrialized, rich, and democratic research participants (so-called WEIRD, Henrich et al. (2010)). The assumption that findings from this group would generalize to all other demographics has proven wrong, and led to a heavily biased corpus of psychological data and research.

Counter measures. Potential counter-measures to demographic selection bias can be simple. Post-stratification is the downsampling of over-represented groups in the training data to even out the distribution until it reflects the actual distribution. Mohammady and Culotta (2014) have shown how existing demographic statistics can be used as supervision. In general, we can use measures to address overfitting or imbalanced data to correct for demographic bias in data. Avoiding biased selections is even better, so when creating new data sets, NLP researchers have been encouraged to provide a **data statement** (Bender and Friedman, 2018). This statement includes various aspects of the data collection process and the underlying demographics. It provides future researchers with a way to assess the effect of any bias they might notice when using the data. As a useful side effect, it also forces us to consider how our data is made up. **data statement**

To prevent the effect of annotation bias, we can use **annotation models** (Hovy et al., 2013; Paun et al., 2018). These models help us find biased annotators, and let us account for the human disagreement between labels. We can use this quantity in the update process of our models (Plank et al., 2014). **annotation models**

1.1.2. *Models.* Another source of biased predictions is the tendency of statistical models for **overamplification**. I.e., the tendency of a model to rely on small differences between subjects to satisfy the objective function and make good predictions. Unchecked, the model can maximize its objective score by amplifying the difference when predicting new data, and creating an imbalance that is much larger than in the original data. Yatskar et al. (2016) have shown that in an image captioning data set, 58% of the captions for pictures of a person in a kitchen mentioned women. A small but **overamplification**

[2] `http://www.socialnomics.net/2010/04/13/over-50-of-the-worlds-population-is--under-30-social-media-on-the-rise/`

noticeable difference, which could be corrected by sampling. However, as Zhao et al. (2017) showed, a standard statistical model trained on this slightly biased data, ended up predicting the gender of a person in a kitchen picture to be a woman in 63% of the cases.

The cost of these false positives seems low. A user might be puzzled or amused when seeing a mislabeled image, or when receiving an email addressing them with the wrong gender. However, relying on models that produce false positives may lead to **bias confirmation** and **overgeneralization**. Some of these false positives might amuse us, but would we accept them if the system predicted sexual orientation and religious views, rather than age and gender? For any text prediction task, this is just a matter of changing the target variable and finding some data.

bias confirmation
overgeneralization

Another problem of overamplification is the proliferation of stereotypes. Rudinger et al. (2018) found that coreference resolution systems (which link a pronoun to the noun they refer to) were biased by gender. In the sentence, "The surgeon could not operate on her patient: it was her son," the model does not link "surgeon" and "her." The cause of this inability is presumably biased training data, but the effect feeds into gender stereotypes. Similarly, Kiritchenko and Mohammad (2018) showed that sentiment analysis models changed their scores for the same sentences when a single word was replaced. E.g., by replacing a female with a male pronoun, or replacing a typically "white" name with a typically "black" name. Both "She/He made me feel afraid" and "I made Heather/Latisha feel angry" result in higher scores for the second case.

Counter measures. To address the overgeneralization of models, we can ask ourselves "would a false answer be worse than no answer?" Instead of taking a *tertium non datur* approach to classification, where a model has to (and will) produce *some* answer, we can use dummy variables that say "unknown." We can also use measures such as error weighting, which incur a higher penalty if the model makes mistakes on the smaller class. Alternatively, we can use confidence thresholds below which we do not assign a label.

adversarial learning

Recently, (Li et al., 2018) have shown that **adversarial learning** (a specialized architecture in neural networks) can reduce the effect of predictive biases. It even helps to improve performance of the models!

1.1.3. *Research Design.* As we have seen in previous chapters (and as this book is further contributing to), most NLP research is on English, and in English. It generally focuses on Indo-European data/text sources, rather than on small languages from other language groups, for example in Asia or Africa. Even if there is a potential wealth of data available from other languages, most NLP tools skew towards English (Schnoebelen, 2013; Munro, 2013).

underexposure

This **underexposure** of other languages creates an imbalance in the available amounts of labeled data. It also reproduces itself. Because most of the existing labeled data covers only few languages, most existing research focuses on those languages. Consequently, this research then creates even more resources for those languages. This

dynamic makes new research on smaller languages more difficult, and it naturally directs new researchers towards the existing languages. The focus on English may therefore be self-reinforcing. The existence of off-the-shelf tools for English makes it easy to try new ideas in English. It requires a much higher startup cost to explore other languages, in terms of data annotation, basic analysis models, and other resources.

There is little in the way of semantic or syntactic resources for many languages. In a random sample of Tweets from 2013, we found 31 different languages. There were no treebanks for 11 of them, and even fewer semantically annotated resources like WordNets.[3] Consequently, researchers are less likely to work on those languages.

Conversely, the prevalence of resources for English has created an **overexposure** to this variety, even though both morphology and syntax of English are global outliers. The overexposure to English (as well as to certain research areas or methods) creates a bias described by the **availability heuristic** (Tversky and Kahneman, 1973). If we can recall something more easily, we infer that this thing must be more important, bigger, better, more dangerous, etc. For instance, people estimate the size of cities they recognize to be larger than that of unknown cities (Goldstein and Gigerenzer, 2002). The same holds for languages, methods, and topics we research. Would we have focused on n-gram models to the same extent if English was as morphologically complex as, say, Finnish?

overexposure

availability heuristic

Lately, there are many approaches to develop multi-lingual and cross-lingual NLP tools for linguistic outliers. However, there are simply more commercial incentives to overexpose English, rather than other languages. Even though other languages are equally fascinating from a linguistic and cultural point of view, English is one of the most widely spoken languages. It is therefore also the biggest market for NLP tools.

Overexposure can also create or feed into existing biases. If research repeatedly found that the language of a particular demographic group was harder to process. This research could create a situation where this group was perceived to be more difficult, especially in the presence of pre-existing biases. The confirmation of biases through the gendered use of language, for example, has been cited as being at the core of second and third wave feminism (Mills, 2012). Overexposure thus creates biases which can lead to discrimination. To some extent, the frantic public discussion on the dangers of AI is a result of overexposure (Sunstein, 2004).

There are no easy solutions to this problem, which might only become apparent in hindsight. It can help to ask ourselves counterfactuals: Would I research this is if the data wasn't as easily available? Would my finding still hold on another language? We can also try to assess whether the research direction of a project feeds into existing biases, or whether it overexposes certain groups.

1.2. Privacy. In the wake of the Cambridge Analytica scandal, it has become apparent that our data is not as secret and private as we would like to think. In the wake

[3]Thanks to Barbara Plank for the analysis!

of this scandal, the European Parliament has enacted a law designed to protect privacy online: the **General Data Protection Regulation** (GDPR).

General Data Protection Regulation

GDPR makes some provisions for research purposes (as opposed to commercial purposes). Still, it does not give out a carte blanche to be negligent with subject data. Non-protected categories can still be predicted for research purposes. Even protected categories are ok to use, as long as they can not be used to identify individual subjects. In other words, if it becomes necessary to estimate the overall prevalence of gender or sexual orientation in the data, we can use models to infer these in aggregate. It is, however, not ok to use them to profile individual subjects.

Counter measures. Protecting privacy can be helped by keeping data sources separate from each other. To still be able to learn from all of these sources, we can use techniques called **federated learning** (Konečnỳ et al., 2016). Coavoux et al. (2018) have shown that neural network architecture choices can help to protect the privacy of users. Still, there is also cautious evidence that this protection might not be as bullet-proof as we might hope (Elazar and Goldberg, 2018).

federated learning

1.3. Normative vs. Descriptive Ethics. Biased models and data sets are a nuisance when used to train a classifier. However, they can also offer a window into the nature of society. This property illustrates an interesting distinction between **normative ethics** and **descriptive ethics**: Because we use language to express opinions, and because word embeddings capture semantic similarity, they also capture how much writers associate two terms. This association is reflected in the fact that word embeddings show a high similarity between "woman" and "homemaker", and between "man" and "programmer". When biased word embeddings are used as input for predictive models, the inherent bias is clearly negative (Bolukbasi et al., 2016), and therefore normatively wrong for many applications (e.g., for reviewing job candidates, where ideally, we would want all genders or ethnicities equally associated with all jobs). However, several social science studies quickly picked up on the insight provided by the biases contained in word embeddings. Works by Garg et al. (2018) and Kozlowski et al. (2018) have shown that we can use precisely this property of word embeddings to study evolving societal attitudes about gender roles and ethnic stereotypes, by measuring the distance between certain sets of words in different decades. Similarly, Bhatia (2017) has shown that this property of word embeddings can be used to measure people's psychological biases and attitudes towards making certain decisions. This bias is therefore descriptively correct.

normative ethics
descriptive ethics

In contrast, Google Translate used "he" to translate the gender-neutral Turkish pronoun "o" as referring to a doctor, but used "she" when referring to a nurse. This translation is both normatively and descriptively wrong. And while it is normatively wrong that Google search autocompletes the query "why are american" with "so fat", it is also descriptively insightful. At least as far as stereotypes are concerned.

1.4. Dual Use. The ethics philosopher Hans Jonas has cautioned that any technology that is possible will also be used for both good and for bad (Jonas, 1984). Even if we do not intend any harm in our experiments, biases in them can still have unintended

consequences that negatively affect people's lives. The most well-known (and extreme) example is physics and the atom bomb. Physicists had to confront the fact that some of their most well-intentioned findings could be (and ultimately were) used to kill.

While in no way as extreme, text analysis techniques can have mixed outcomes as well. On the one hand, they can vastly improve search and educational applications (Tetreault et al., 2015), but they can also re-enforce prescriptive linguistic norms when they work poorly for non-standard language. Stylometric analysis can shed light on the provenance of historic texts (Mosteller and Wallace, 1963) and aid forensic analysis of extortion letters, but it can also endanger the anonymity of political dissenters. Text classification approaches can help decode slang and hidden messages (Huang et al., 2013), but have the potential to be used for censorship and suppression. At the same time, NLP can also help to uncover such restrictions (Bamman et al., 2012). Hovy (2016) shows that simple NLP techniques can be used to both detect fake reviews, but also to generate them in the first place.

All these examples indicate that we should become more aware of the way other people appropriate NLP technology for their purposes. We also need to be mindful that NLP research is and will be used for solely unwanted applications. Automated censorship or the measuring of party-line adherence online to punish dissenters are two examples. Statistical models make both of these uses possible. The unprecedented scale and availability can make the consequences of new technologies hard to gauge.

The examples show that moral considerations go beyond immediate research projects. As a practitioner, we need to be aware of this duality and openly address it. We may not directly be held responsible for the unintended consequences of our research or products, but we should acknowledge how they can enable morally questionable or sensitive practices, raise awareness in our customers, colleagues, and students, and lead the discourse on it in an informed manner. The role of the researcher in such ethical discussions has been pointed out by Rogaway (2015), and the moral obligations for the practicing data scientists in O'Neil (2016).

2. Classification

spam filtering
sentiment analysis
author attribute prediction

Very often, we are interested in predicting whether a document belongs to a particular class of interest: is it an ad or not (**spam filtering**)? Is it positive, negative, or neutral in tone (**sentiment analysis**)? Was it written by an older or a younger person (**author attribute prediction**)? Most of those applications are commercial. Still, predictive approaches based on text are becoming more and more common in social sciences as well. Suppose we can predict from a speech which party the politician belongs to or whether a report outlines human rights. That tells us something about the level of political polarization (Peterson and Spirling, 2018) and evolving human rights standards (Greene et al., 2019). To answer all of these questions, we can use predictive statistical models (or **classifiers**). They have been fitted on training texts to maximize the number of correct predictions on held-out-data.

classifiers

Classification is one of the core machine learning applications, which has gained a lot of attention over the last few years. It can seem complicated and mysterious, but all machine learning is closely related to statistical methods commonly used in social science. One of the most common (and valuable) classification algorithms is **logistic regression**, the same method used to compute the fit between a set of independent variables and a binary dependent variable in social sciences. The algorithm is the same, but there are two main differences.

logistic regression

The first set of differences is terminological: instead of independent variables, machine learning talks about **features**; instead of dependent variables, it uses **targets**, whose values are called **classes** or **labels**; the coefficients (or betas) of the algorithm are referred to as **parameters** or **weights**. We will use the machine learning terminology depending here, with occasional translations.

features
targets
classes
labels
parameters
weights

In general, we want to represent the data we work with as **vectors** (i.e., lists of numbers) and **matrices** (i.e., a list of vectors). These formats have the advantage that we can apply all kinds of linear algebra techniques to implement our algorithms, which are theoretically well-founded and computationally fast. [4] We represent each example as a vectors X_i, and stack them together to form a **data matrix** $\mathbf{X}$. Each feature is one column in the matrix. The labels for all the instances are stored in a vector y. Formally, each training instance is a tuple (X_i, y_i) where X_i is a D-dimensional vector of indicators or real numbers describing the input, and $y \in K$ is the correct output class from a set of possible class labels, K. We also refer to the correct labels as **gold labels** or **ground truth**.

vectors
matrices
data matrix
gold labels
ground truth

So a data point is characterized by the values of all its features. Visually, we can imagine each feature as a dimension in a grid by placing it at a point in space. Ideally, we find dimensions representing the data so that the classes form clusters that are easily

[4] Most machine learning, natural language processing, and AI boils down to a series of linear algebra operations. So it would be technically correct to replace all of them with "matrix multiplication", but it does not make for nearly as exciting headlines.

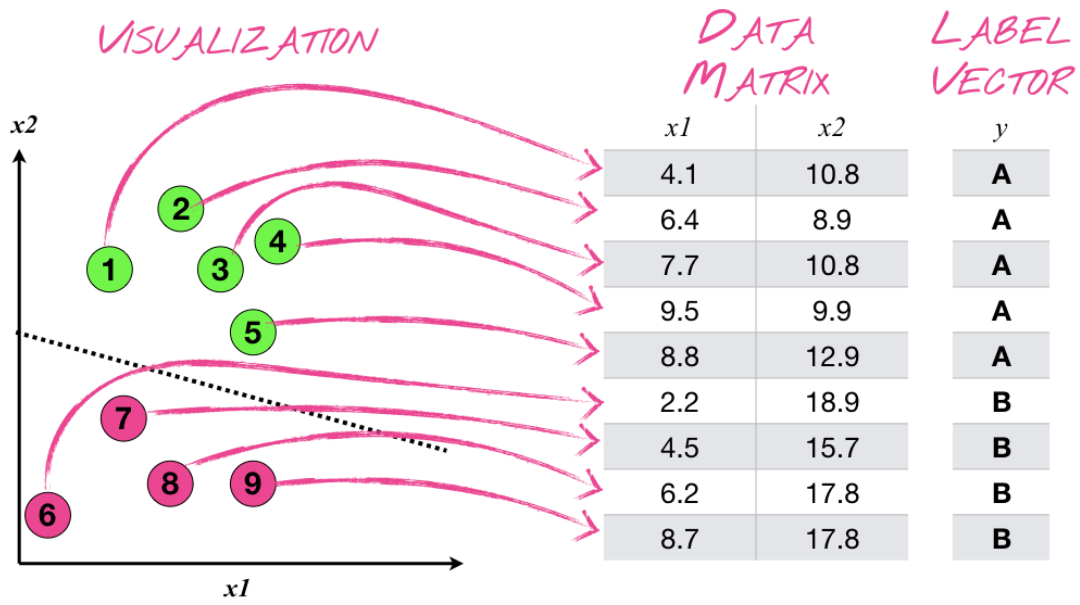


FIGURE 1. Example of a two-class problem with the data points plotted along their two feature dimensions, colored by class label, and separated by a decision boundary (left), and the corresponding data matrix $\mathbf{X}$ with two columns and the label vector y

separable with a line (or, in more than two dimensions, a hyperplane). See Figure 1 for an example.

Independent of the classifier, we can write all of these approaches as the familiar equation

$$y = f(X; \theta, b)$$

That is, the output value y is a function $f(\cdot)$ of the input X, parametrized by a set of parameters, θ, and a **bias term** b. Depending on the algorithm, θ can be weights, probabilities, or activation functions. We will also see several ways to use b to make our estimates better in Section 7. In the case of a linear model, like Logistic Regression, $f(X)$ would be

bias term,

$$\theta \cdot \mathbf{X} + b$$

We multiply each value of X with the corresponding value in θ, and sum up the results. We add b and threshold the result to get the output. We can visualize the "matrix view" of this operation as in Figure 2 (we are ignoring b for the moment).

The output of this binary model is a value between 0 and 1. By checking against a threshold (typically 0.5), we can decide which of the two classes to assign. Once we have fitted our θ vector of weights as in the example above, we can apply it to new

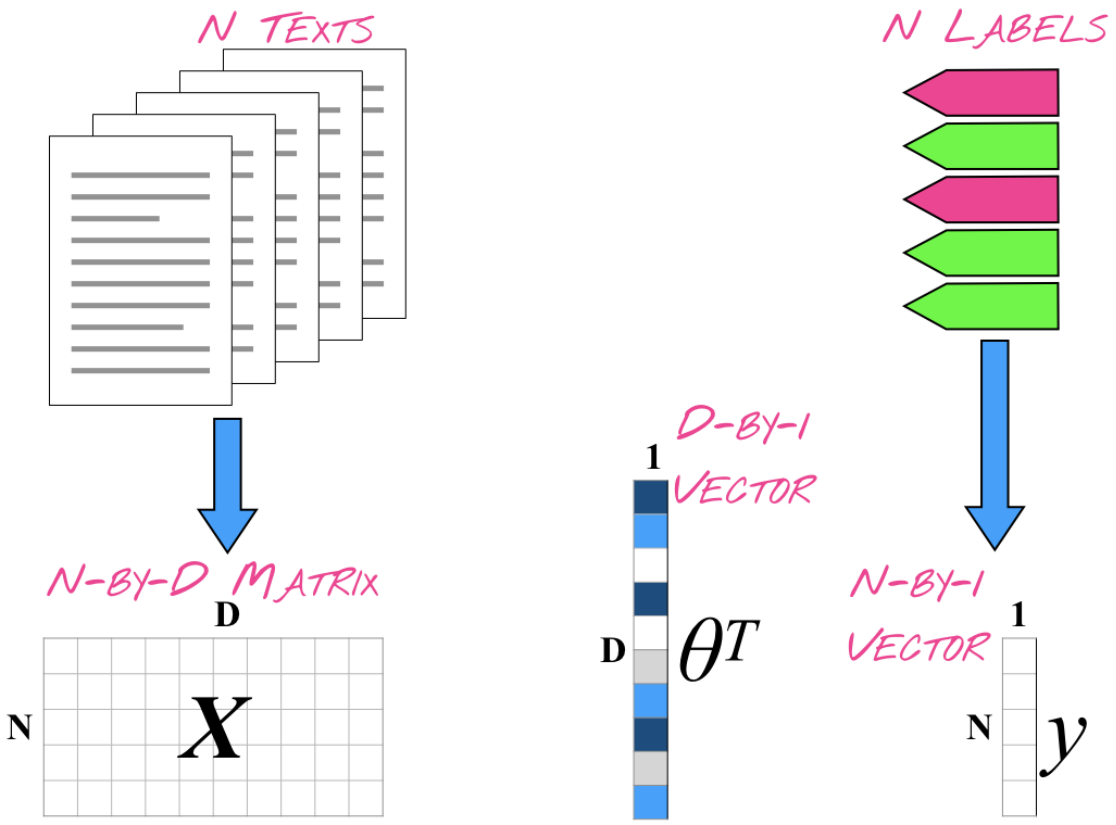


FIGURE 2. Matrix representation of fitting a classifier: we learn vector θ to transform matrix $\mathbf{X}$ into vector y

data $\mathbf{Z}$ (provided it has the same number of columns). The result are predictions of our model. To distinguish the gold labels y from these predictions, we use a hat over the predictions: $\hat{y}$. Mathematically, we can say this as

$$\hat{y} = \theta \cdot \mathbf{Z} + b$$

And graphically, we can represent it as in Figure 3 (again ignoring b for now).

There are many machine learning classifiers. Logistic Regression is the most common, but it can sometimes be good to use Support Vector Machines (Section 8.1), Naive Bayes (Section 8.2), or (Feed-Forward) Neural Networks (Section 13). We will cover all of these. These methods differ only in the ways they weigh and combine the input. However, the principle is always the same: find a pattern of input features, weigh their evidence, and map that to the output target class. You do this yourself when you scan a possible spam email: does it contain features like "dear friend", "amazing offer", or "FREE"? How likely is each of those to be spam? If it contains only one, it might be ham, but if there are too many red flags, it is probably spam.

multi-class

Suppose we have more than two classes (i.e., a **multi-class** problem), like in

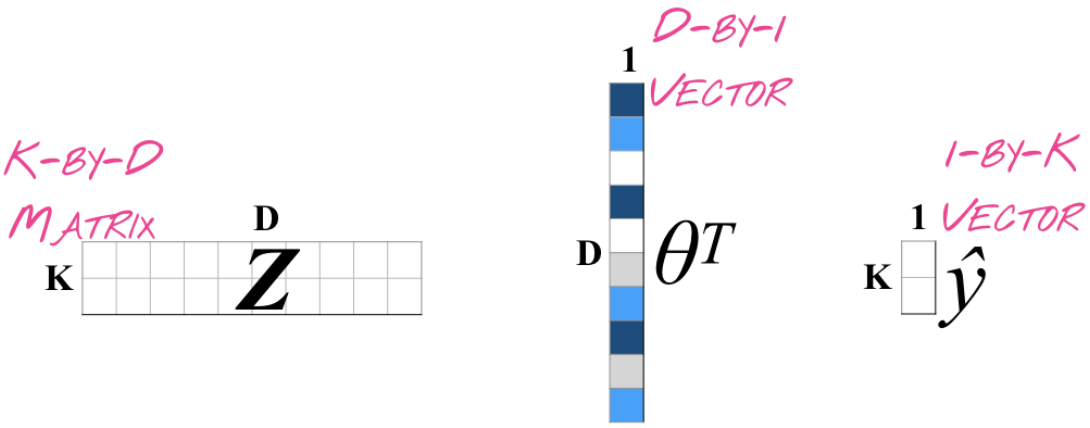


FIGURE 3. Matrix view of prediction: multiplying a learned weight vector θ with a new data matrix $\mathbf{Z}$ to get the prediction vector $\hat{y}$

sentiment analysis, where we want to choose between three outcomes. In that case, we have two options. We can train a separate binary model for each class (each learns to distinguish instances of that class from all others) and then choose the one with the highest confidence. If we learn a multi-class problem in `sklearn`, the latter is what Python does under the hood. Alternatively, we can train a model that predicts a probability distribution over all the possible output values and then choose the one with the highest value. This distribution is called a **softmax** (technically, we exponentiate each value before normalizing, which exaggerates initial differences more than "regular" normalization).

softmax

The second difference between logistic regression as a classifier in machine learning and as a regression method in social science is its application: **prediction vs. explanation**. In social science, we fit a model to the data to find an explanation. We examine the model coefficients to find an explanatory causal correlation between the independent variables and the dependent variable. Very rarely do we test the predictive power of this model on new data. In machine learning, in contrast, the objective is only to perform well on new data. After fitting, we freeze the coefficients and measure how well the fitted model predicts new, held-out data. These two camps need not be an either-or: several researchers have shown (Shmueli et al., 2010; Hofman et al., 2017; Yarkoni and Westfall, 2017) that using machine learning methods to test the predictive accuracy of a model is a good robustness test. Both fields try to reverse-engineer the underlying data generation process. Ultimately, social science theory is also about predicting how systems (people, markets, firms) will behave under similar conditions in the future. The critical difference is that every causal model is necessarily also a good predictive model. If we know the causal relation between elements, we can predict how they will develop. The inverse, however, is not necessarily true. A good predictive model might

prediction vs. explanation

capture aspects that have nothing to do with causality (we will see this when we come to overfitting in Section 7).

2.1. Checklist Text Classification. Here is a step-by-step checklist on how to classify text data and code to implement one possible solution. We will elaborate on the individual steps in the following sections.

(1) label *at least* 2000 instances in your data set
(2) preprocess the text of *all* instances in your data (labeled and unlabeled)
(3) read in the labeled instances and their labels
(4) transform the texts into feature vectors using `TfidfVectorizer` or embeddings
(5) optional: select the top N features (where N is smaller than the number of labeled instances)
(6) fit a classifier
(7) use 5-fold CV to find the best regularization parameter, top N feature selection, and potentially feature generation and preprocessing steps

```
from sklearn.linear_model import LogisticRegression
from sklearn.dummy import DummyClassifier
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import GridSearchCV
from sklearn.calibration import CalibratedClassifierCV

vectorizer = TfidfVectorizer(ngram_range=(1,2),
                             min_df=0.001,
                             max_df=0.75,
                             stop_words='english')

X = vectorizer.fit_transform(data['clean_text'])
y = data['output']

print(X.shape, y.shape)

# get baseline performance
most_frequent = DummyClassifier(strategy='most_frequent')
print(cross_val_score(most_frequent, X, y=y, cv=5, n_jobs=-1, scoring
    ="f1_micro").mean())

# fine-tune classifier
base_clf = CalibratedClassifierCV(cv=5,
    base_estimator=LogisticRegression(n_jobs=-1,
                                      solver='lbfgs'
                                     )
)
```

```
param_grid = {'base_estimator__C': [50, 20, 10, 1.0, 0.5, 0.1, 0.05,
    0.01],
              'base_estimator__class_weight': ['balanced', 'auto']
             }
search = GridSearchCV(base_clf, param_grid, cv=5, scoring='f1_micro')
search.fit(X, y)

# use best classifier to get performance estimate
clf = search.best_estimator_.base_estimator
print(cross_val_score(clf, X, y=y, cv=5, n_jobs=-1, scoring="f1_micro
    ").mean())
```

CODE 46. Training a text classifier.

```
from sklearn.feature_selection import SelectKBest
from sklearn.feature_selection import chi2
from sklearn.pipeline import Pipeline

# set up the sequence
pipe = Pipeline([
    ('reduce_dim', 'passthrough'),
    ('classifier', clf)
])

# specify selection range
N_FEATURES = [1800, 1500, 1000, 500, 300]
param_grid = [
    {
        'reduce_dim': [SelectKBest(chi2)],
        'reduce_dim__k': N_FEATURES
    },
]

# fit the model to different feature sets
grid = GridSearchCV(pipe, n_jobs=1, param_grid=param_grid, cv=5,
    scoring='f1_micro')
grid.fit(X, y)

# save the best selector
selector = grid.best_params_['reduce_dim']
X_sel = selector.transform(X)

# refit classifier on entire, dimensionality-reduced data set
clf.fit(X_sel, y)

```

```
cv_reg = cross_val_score(clf, X_sel, y=y, cv=5, n_jobs=-1, scoring="
    f1_micro")
print("5-CV on train: {}".format(cv_reg.mean()))
```

CODE 47. Feature selection.

Once you are satisfied with the results and want to apply the classifier to new data:

(1) read in the (unlabeled) new data
(2) use the `TfidfVectorizer` from 5. above to transform the new instances into vectors
(3) use the `SelectKBest` selector from 6. above to get the top N features
(4) use the classifier from 7. above to predict the labels for the new data
(5) save the predicted labels or probabilities to your database or an Excel file

```
# read in new data set
# transform text into word counts
# IMPORTANT: use the same vectorizer we fit on training data to
    create vectors!
Z = vectorizer.transform(new_data['clean_text'])

# select features for new data
Z_sel = selector.transform(Z)

# use best classifier to predict labels
predictions = clf.predict(Z_sel)
```

CODE 48. Apply classifier to new data.

3. Labels

The process of assigning labels to documents is typically called *coding* in social sciences. However, that term is already reserved for writing code in computational sciences, so the general term used here is **annotations**. That is not the only difference concerning labels between the fields.

annotations

In social science, texts are typically annotated by a single person. However, people have quirks and biases. They might be tired and inattentive when they annotate a batch, or they might not be too involved in the task and do it mainly for the money. Annotation decisions made by the coders of our data introduce **annotation bias**. Sometimes, biases can arise because the annotators were not familiar with the language or style in the data, using their own understanding to label the examples instead Sap et al. (2019).

annotation bias

Even with the best intentions, people do make mistakes and are inconsistent at times. In NLP, documents are typically annotated by multiple people, often on crowdsourcing platforms. That redundant annotation allows us to aggregate the answers and find a more robust, unbiased label. It also allows us to determine 1) how reliable annotators are, 2) how difficult specific texts were to label, and 3) how

much disagreement exists between annotators. NLP papers will typically report the **inter-annotator agreement** and how they aggregated the results. Because multiple annotations are more expensive, papers sometimes measure agreement only on a subset of the data. While theoretically cheaper but equivalent to trained annotators in quality (Snow et al., 2008), it is not always clear whether the demographic makeup of these anonymous workers is representative (Pavlick et al., 2014). Crowdsourcing also raises ethical questions about worker payment and fairness (Fort et al., 2011).

inter-annotator agreement

To prevent the effect of annotation bias, we can use **annotation models** (Hovy et al., 2013; Passonneau and Carpenter, 2014; Paun et al., 2018), i.e., Bayesian models that infer the most likely answer and the reliability of each annotator. These models help us find biased annotators and let us account for the human disagreement between labels. Implementations of such models are available as software (`https://github.com/dirkhovy/mace`) or web service (`https://mace.unibocconi.it/`). We can even use this quantity in the update-process of our models (Plank et al., 2014).

annotation models

We are working with matrices and linear algebra operations. So internally, we cannot use the actual class names (e.g., "informative", "emotional", "unrelated") of our output labels. `Sklearn` will translate all our text labels into numbers to allow us to do linear algebra. If, however, we want to do this ourselves, `sklearn`, lets us translate any collection of labels into number starting from 0, using the `LabelEncoder`:

```
from sklearn.preprocessing import LabelEncoder

# transform labels into numbers
labels2numbers = LabelEncoder()
y = labels2numbers.fit_transform(labels)
```

CODE 49. Translating labels into numbers.

If we use this numerical representation (again, we do not have to), the output of classifier in `sklearn` will also be numbers. Since this is harder to interpret, we have to translate the resulting numbers back into class labels:

```
# translate numbers back into original labels
predicted_labels = labels2numbers.inverse_transform(predictions)
```

CODE 50. Translating predictions back.

4. Train-Dev-Test

When we see a new message in our inbox, we usually know pretty quickly that it is spam. There are indicators in the text that help us figure that out, based on our experience with previous spam messages. Classifiers work very similarly.

training

We assume that the output label is related to the input via some function that we need to find by fitting several parameters. This process is called **training**, and it corresponds to our experience with previous spam messages. The training goal is to let the machine find repeated patterns in the data that can be used to estimate the correct output for a new, held-out input example. This new sample is called the **test set**, and in our spam example, it corresponds to a new message in our inbox. We are interested in achieving the highest possible score of some appropriate **performance** metric on the held-out test data.

test set

performance

```
from sklearn.linear_model import LogisticRegression

classifier = LogisticRegression()

classifier.fit(X, y)

predictions = classifier.predict(Z)
```

CODE 51. Example of a fitting a simple Logistic Regression classifier and applying it to new data to predict its output.

We will later see how to improve on this basic classifier (Sections 7 and 8). To do that, however, we first need to discuss how to measure these improvements. That includes both data handling and metrics.

We have already briefly discussed the role of the held-out sample for prediction, i.e., the test data on which we measure our classifier's performance. But where does that test data come from? If we simply take a new unlabeled text, how can we measure the predictive performance? If we don't know the correct labels for those new data points, what do we compare the predictions against? To simulate encountering new, unseen instances, we divide our data set of (X_i, y_i) pairs into three parts before fitting our model.

training set

test set

The first and largest part of these three subsets is the **training set**, which we will use to fit our model. We also set aside some of the data that we will use as **test set**, and do not use or look at it until we are content with our model. We pretend this data set does not exist. Therefore, we can not use the test set to collect words for our vocabulary, look at the distribution of labels, or collect any other statistics.

There is one problem with this setup, though. Models usually have several parameters we can set. We will need to find the setting that works best for our particular data sample. To know whether a specific parameter setting works, we need to evaluate it on held-out data. However, if we adjust the parameters often enough and measure performance on the test data after every time, we will end up with excellent performance. But we essentially cheated: we chose the parameters that made it happen! In real life, we could not know these parameters. We can only measure performance on the test set once!

So how *do* we measure the effect a parameter change has on predictive performance if we cannot use the test set? Instead of the test set, we can use another data set, which hopefully approximates the test set. We can use this third data set to see how changing the model parameters affects performance on *a* held-out data set. This second held-out data set, which we use to tune our model parameters, is called the **development set** (or dev set for short).

development set

You can think about this as preparing for an important presentation to funders. You can only present to them once (i.e., they are your test set), but you might want to tweak a few things in the presentation (those are your parameters). Since you cannot do a mock presentation with the funders, you ask some colleagues and friends to act as a stand-in. They are your development set. Obviously, the more similar your practice audience is to the funders, the better you will be able to gauge whether you are on the right track.

For the same reason, we want our development and test set to be as similar to each other as possible, i.e., equal size, representative distributions over the outcomes, and large enough to compute robust statistics. To fulfill these conditions, both sets need to be large enough. If you only had ten documents in your dev set, your results will not be representative. It is hard to put a fixed number on this, but at the very least, you should have several hundred instances for both of these data sets.

At the same time, you want your training set to be large enough to fit the model correctly. If you had too few documents, you might never see enough combinations to set all the model parameters properly. Again, as a rough guideline, you should have thousands of instances. For a binary classification problem, 1500 to 2000 instances are a good start, but more is always better. We will see several ways to improve performance later (Sections 7 and 8). However, the best way is still to add more training data. Common ratios of training:development:test data size are $80{:}10{:}10$, $70{:}15{:}15$ or $60{:}20{:}20$, depending on which ration best satisfies the above conditions.

4.1. Cross-Validation. That is all fine and well, you might say, but I only have a data set of 1500 documents. I cannot set aside some of them as a test set I use only once, or I reduce the amount of data I have to fit the data too much. You have a good point.

For this case, you can use k**-fold cross-validation**. In k-fold cross-validation, we simulate new data by dividing the data into k subsets, fitting the k model on $k-1$ subsets of the data, and evaluating on the k^th subset. See Figure 4 for an example illustration with three folds. Each document has to end up in the held-out part once. In the end, we have performance scores from k models. The average over these scores tells us how well the model would work on new data.

k-fold cross-validation

This approach has several advantages: we can measure the performance of the entire data set, simulate different conditions, and get a much more robust estimate. Again, choosing k depends on the resulting training size and the test portions for each fold. k should be small enough to guarantee a decent-sized test set in each fold but large enough to make the training part sufficient. Common values for k are 3, 5, or 10. In the extreme, if $k = N$, the size of our vocabulary, we fit one model and test it separately for each document in our corpus. This version is also called **Leave-One-Out**

Leave-One-Out cross-validation*(LOO)*

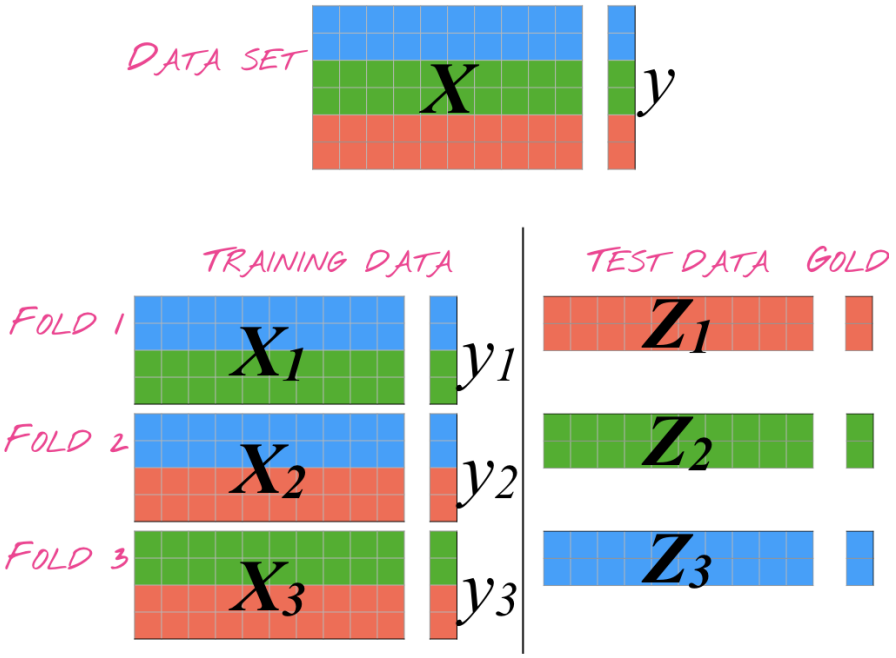


FIGURE 4. Schematic of 3-fold cross-validation

cross-validation.

If we want to have a dev set to tune each of the k models, we can further split each of the $k-1$ parts of each fold's training portion. In that case, we train on $k-2$ parts and tune and test on k parts each.

In Python, we can get stratified subsets of the data (meaning that the label distribution is the same in each set):

```
from sklearn.model_selection import StratifiedKFold

skf = StratifiedKFold(n_splits=3)
for train_ids, test_ids in skf.split(X, y):
    fold_X_train = X[train_ids]
    fold_y_train = y[train_ids]

    fold_X_test = X[test_ids]
    fold_y_test = y[test_ids]
```

CODE 52. Example of stratified cross-validation.

5. Performance Metrics

How good is your spam filter? If you get a spam email that was not caught, you might say "not great". The occasional spam email is annoying, but how many does it actually catch? Clearly, we need to be able to measure these things somehow. If someone handed you a classifier and said it was 70% accurate, would you use it? What about 80% accuracy? 90%?

In explanatory models, we usually are only interested in the fit of the model to the data. That quantity is typically the r^2 score. While r^2 is *a* performance metric, it is difficult to generalize it to held-out data: how well does a line fit points that we do not know yet? In prediction, we are less interested in the fit but in how well the classifier will predict future cases. We want to measure for how many of the documents in our test data a model produces the correct answer. Depending on how many classes we have and how balanced our data set is for these classes' distribution, there are different measures we can use. For all of them, we will use the three basic counts: **true positives** (TP the model prediction and the actual label are the same), **false positives** (FP, the model predicts an instance to be of a specific class, but it is not), **true negatives** (TN, the model correctly recognizes an instance *not* to be of a specific class), and **false negatives** (FN, the model fails to recognize that an instance belongs to a specific class).

true positives

false positives

true negatives

false negatives

Measures of fit and performance measures need not be mutually exclusive. Yarkoni and Westfall (2017) argued that performance measures can and should be used as an additional measure of robustness. They can also stand in as proxies for other values. Peterson and Spirling (2018) have shown that a higher predictive accuracy of party affiliation correlates with periods of higher polarization. The more polarized their speeches, the easier it is to tell politicians apart.

The most straightforward measure of this performance is **accuracy**. We simply divide the number of true positives by the total number of instances, N.

accuracy

$$acc = \frac{TP + TN}{N}$$

Accuracy is a helpful measure, but somewhat less helpful when our labels are unbalanced: in a data set where 90% of all labels are class `A`, and 5% each are class `B` and `C`, a system that always predicts `A` would get an accuracy of 0.9 (all performance measures are usually given as floating-point values between 0 and 1. Sometimes, people also report accuracy as a percentage, though).

When our label distribution is more skewed, we need different measures than accuracy. **Precision** measures how many of our model's predictions were correct. We divide the number of true positives by the number of all positives.

Precision

$$prec = \frac{TP}{TP + FP}$$

In the case where we have only two equally frequent classes, precision is the same as accuracy.

Recall

Recall measures how many of the positive examples in the data our model managed to find. We divide the number of true positives by the sum of true positives (the instances our model got right) and false negatives (the instances our model *should* have gotten right but did not).

$$rec = \frac{TP}{TP + FN}$$

A model that classified everything as, say, *positive*, would get a perfect recall for that class. It does, after all, find *all* true positives while not producing any false negatives. However, such a model would obviously be useless since its precision is terrible.

F1 score

To get a single number to detect the case above, we want to balance precision and recall against each other. The **F1 score** does precisely that by taking the harmonic mean between precision and recall.

$$F_1 = 2 \times \frac{prec \times rec}{prec + rec}$$

Let's look at a number of examples (see Table 1). We will use the following data, from a hypothetical classifier that identifies animals (`1`) and natural objects (`0`).

X	y	$\hat{y}$	if target=1	if target=0
frog	1	1	TP	TN
deer	1	1	TP	TN
wolf	1	1	TP	TN
dog	1	1	TP	TN
bear	1	1	TP	TN
fish	1	1	TP	TN
bird	1	0	FN	FP
cat	1	0	FN	FP
stone	0	1	FP	FN
tree	0	0	TN	TP

TABLE 1. Example output of a classifier and their valuation for different target classes.

If we are interested in animals, our target label is `1`, and we get the following metrics from the measures in the 4th column:

$$acc = \frac{7}{10} = 0.7$$

$$prec = \frac{6}{7} = 0.86$$
$$rec = \frac{6}{8} = 0.75$$
$$F_1 = 0.81$$

If instead our target label is `0`, we get the following metrics:

$$acc = \frac{7}{10} = 0.7$$
$$prec = \frac{1}{3} = 0.33$$
$$rec = \frac{1}{2} = 0.5$$
$$F_1 = 0.4$$

Note that accuracy is not affected by how many classes we have or what class we focus on: we only check how often the prediction and the gold data match. Which classes the matching instances have does not matter. For precision, recall, and F1, we need to calculate the score for each class separately. If we have *positive*, *negative*, and *neutral* as labels, and our target class is *positive*, either of the others is a false negative.

Even if we have several classes and compute precision, recall, and F1 separately, we might still want to have an overall metric. There are two ways of doing this: **micro-averaging** and **macro-averaging**.

micro-averaging
macro-averaging

In micro-averaging, we use the raw counts of the positives and negatives, and add them up. This gives us an average that is weighted by the class size, as larger classes have a greater influence on the outcome.

$$acc_{micro} = 7/10 + 7/10 = 14/20 = 0.7$$
$$prec_{micro} = 6/7 + 1/3 = 7/10 = 0.7$$
$$rec_{micro} = 6/8 + 1/2 = 7/10 = 0.7$$
$$F_{1micro} = 0.7$$

In macro-averaging, we first compute the individual scores for each class, and then take the average over them. This gives us an average that weighs all classes equally.

$$acc_{macro} = (0.7 + 0.7)/2 = 0.7$$
$$prec_{macro} = (0.86 + 0.33)/2 = 0.6$$
$$rec_{macro} = (0.5 + 0.75)/2 = 0.63$$
$$F_{1macro} = 0.61$$

Which averaging we should use very much depends on the goal we have and how much importance we attach to small classes. There are good reasons to use either scheme.

In Python, we can get all of these performance measures by comparing our classifier's predictions against the gold labels of the held-out data:

```
from sklearn.metrics import f1_score, precision_score, recall_score

prec = precision_score(gold, prediction, average="micro")
rec = recall_score(gold, prediction, average="micro")
f1 = f1_score(gold, prediction, average="micro")
```

CODE 53. Examples of evaluation metrics.

If we do not have a dedicated heldout set, we can get the average score via cross-validation:

```
from sklearn.model_selection import cross_val_score

prec_cv = cross_val_score(clf, X, y=y, cv=5, n_jobs=-1, scoring="
    precision_micro")
rec_cv = cross_val_score(clf, X, y=y, cv=5, n_jobs=-1, scoring="
    recall_micro")
f1_cv = cross_val_score(clf, X, y=y, cv=5, n_jobs=-1, scoring="
    f1_micro")
```

CODE 54. Example of micro-averaged evaluation metrics in cross-validation.

Instead of micro-averaging, we can use macro-averaging by changing the `scoring` parameter to `M_macro`, where `M` is any of the above metrics. For a full list of all the available cross-validation scoring functions, you can use this code:

```
import sklearn.metrics as mx
sorted(mx.SCORERS.keys())
```

CODE 55. Available evaluation metrics for cross-validation.

6. Comparison and Significance Testing

When we train a classifier, we want to use it in the future on unseen data. We have seen how we can measure and predict that performance and guard ourselves against overfitting. However, while performance measures give us a good sense of the model's capabilities, it does not tell us anything about the classification task's difficulty. Imagine a colleague tells you about their classifier that got an F1 score of 0.9 in distinguishing advice from non-advice in a set of social media posts. It turns out, though, that less than 1% of the observations are advice. So what did the classifier learn? Is it really finding those 1% of advice cases, or has it "given up" and just classified everything as none-advice? If the data is extremely imbalanced (where most examples are from the same class), a 0.9 F1 score performance does not mean much.

baseline

To get a sense of the possible headroom, researchers often run a very simple **baseline**

first. This baseline can be an "algorithm" that always predicts the most frequent class label (**majority baseline**) or another simple model. With non-linear models (Section 13), we will see that it makes sense to compare against a linear model's baseline. A model that does not even outperform the baseline is probably not on the right way. After all, it does not make sense to use a more complicated model where a simple one suffices.

majority baseline

`Sklearn` provides a simple implementation of the most-frequent-label classifier. Since the classifier always returns the most frequent label for everything, we do not even have to use cross-validation. Instead, we can look at the performance of the entire data.

```
from sklearn.dummy import DummyClassifier
from sklearn.metrics import f1_score

most_frequent = DummyClassifier(strategy='most_frequent')
most_frequent.fit(X, y)
most_frequent_predictions = most_frequent.predict(X)
print(f1_score(y, most_frequent_predictions, average='micro'))
```

CODE 56. Running a baseline classifier that predicts the most frequent label.

What if our model outperforms the baseline by some margin? Does that mean another person will see the same results when running the model on their data? Are the improvements over the baseline that we reported a fluke, or do they generalize? This question is essentially at the heart of **statistical significance** tests. [5]

statistical significance

There are several statistical significance tests out there. However, it requires some knowledge to pick the right one for the task at hand (Berg-Kirkpatrick et al., 2012). For many NLP problems, though, bootstrap sampling is a valid and interpretable test. The intuition behind it is simple: when we compare the results of our model against a baseline, we only have one comparison, which might be biased. The result could very well be due to the particular composition of the data and disappear on a different data set. The best way to address this concern is to test on more data sets. However, if we do not have other samples, we can approximate the effect by simulation. We create different data sets by repeatedly sampling from the data with replacement (which is called **bootstrapping**). We compare both baseline and model performance on each of those samples and record how often that difference is more extreme than on the complete data. The **central limit theorem** says that repeated measures taken from a population follow a normal distribution. Think of a stadium full of people, from which you repeatedly draw five and measure their average height. Most of the time, you will get an average height, and only rarely will you get all the tall people. The average heights will follow a normal distribution. We can use this insight to identify

bootstrapping

central limit theorem

[5] There are some more factors to consider: models are sensitive to domain differences – a model trained on newswire will not work well on Twitter data. But say our user wants to apply the model to the same domain.

samples where the difference between baseline and model is different from the base case. There are two ways to measure this. The first is the number of samples where the difference deviates more than two standard deviations from the original difference between the systems. The other way is to count negative differences (meaning samples where the better system on the base case is actually worse). The latter case might seem more intuitive, but it requires some assumptions. 1) the mean sample difference is the same as the difference on the entire data set, and 2) the distribution over differences is symmetrical. Under those conditions, it is equivalent to the first version. Most NLP measures will fulfill those conditions.

Take the examples in Table 2: on the entire sample, system A is 0.24 points better than system B. If we take enough samples computing the difference between A and B, these differences will follow a normal distribution, and the average difference will be very close to 0.24. We are asking with bootstrap sampling: *in how many of these samples is the difference at least twice as extreme as on the base case*? This setup requires us to put the system with a higher score on the entire data set first. We collect ten samples here (this is only for demonstration purposes, we usually would gather at least 10,000).

	A	B	difference
base	82.13	81.89	0.24
1	81.96	82.03	-0.07
2	81.86	82.61	-0.75
3	81.70	81.44	0.26
4	82.42	82.77	-0.35
5	81.89	81.06	0.83
6	81.39	81.24	0.15
7	81.96	81.58	0.37
8	82.57	81.65	0.92
9	82.50	82.67	-0.17
10	83.07	81.84	1.23

TABLE 2. Example performance of two systems on the entire data set (base) and on 10 samples.

Figure 5 shows the distribution over the observed differences. As expected, their average difference is about 0.24, the same as the difference between the systems on the entire data. However, in three cases (samples 5, 8, and 10), the difference is more than twice the base difference. Because the sampled differences follow a normal distribution, and because the mean difference is small, there are some cases where A is actually worse than B. These cases indicate that the observed difference is not significant. The p-value here would be $3/10 = 0.3$. (Again, in a real test, we would use a much higher number

of samples, which would make the result more exact and more reliable). As the base difference increases, there are fewer and fewer extreme cases.

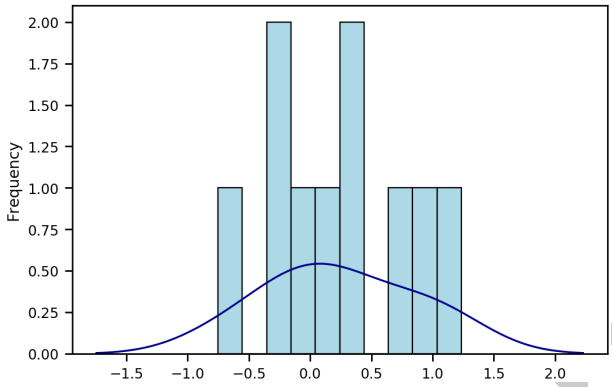


FIGURE 5. Distribution over differences between the two systems on 10 samples

```python
import numpy as np

def bootstrap_sample(system1, system2, gold, samples=10000, score=precision_score):
    """
    compute the proportion of times the performance difference of the
    two systems on a subsample is significantly different from the
    performance on the entire sample
    """
    N = len(gold) # number of instances

    # make sure the two systems have the same number of samples
    assert len(system1) == N and len(system2) == N, 'samples have different lengths'

    # compute performance score on entire sample
    base_score1 = score(gold, system1, average='binary')
    base_score2 = score(gold, system2, average='binary')

    # compute the difference
    basedelta = base_score1 - base_score2
    assert basedelta > 0, 'Wrong system first, system1 needs to be better!'

    system1 = np.array(system1)
    system2 = np.array(system2)
    gold = np.array(gold)
```

```
25
26      p = 0
27      for i in range(samples):
28          # select a subsample, with replacement
29          sample = np.random.choice(N, size=N, replace=True)
30
31      # collect data corresponding to subsample
32          sample1 = system1[sample]
33          sample2 = system2[sample]
34          gold_sample = gold[sample]
35
36      # compute scores on subsample
37          sample_score1 = score(gold_sample, sample1, average='binary')
38          sample_score2 = score(gold_sample, sample2, average='binary')
39          sample_delta = sample_score1 - sample_score2
40
41          # check whether the observed sample difference is at least
42          # twice as large as the base difference
43          if sample_delta > 2*basedelta:
44              p += 1
45
46      return p/samples
```

CODE 57. Bootstrap sampling for statistical significance tests.

7. Overfitting and Regularization

Imagine a (slightly contrived) case where for some reason, most positive training examples in a text classification task contained the word "Tuesday". So, the presence of "Tuesday" is an excellent feature to predict positive instances from the classifier's perspective. But suppose the new instances we encounter in our test set never contain the word "Tuesday". In that case, the model comes up empty-handed when it tries to classify them. It has been **overfitting** to the training data. This issue is similar to memorizing vs. understanding. If we memorize that $2 + 2 = 4$, we might be surprised when we are told that $3 + 1$ is also 4. If, instead, we understand how addition works, we will be much less surprised by future cases.

overfitting

Training (i.e., fitting) a model on too many features can cause the model to memorize parts of the training data. While that is excellent for predicting *those* outcomes correctly, it is entirely useless when encountering new examples.

In most social science scenarios, we only use a small number of independent variables. In machine learning, we often use thousands of them. When working with text, one of the easiest ways to **featurize** the input is to make each observed word type a feature. Depending on the estimate, English has between $100,000$ and $500,000$ words,

featurize

so the number of features quickly becomes enormous. This number of independent variables is frowned upon in social sciences – and for a good reason.

In the extreme case, if we had more independent variables than observations, we can simply find one variable that explains the data. However, it is hard to know whether this explanation has any validity. In machine learning, this situation is also frowned upon, but for different reasons.

In prediction, we would like to prevent the model from perfectly memorizing the training data. If it does, it will not be able to predict new instances accurately. (Remember the example of $2 + 2 = 4$ and $3 + 1 = 4$). This point is where the bias term b we mentioned earlier comes back into it. Choosing a bias that prevents the model from overfitting is called **regularization**. It is always a good idea to regularize the models, even if we do not want to use them for prediction.

regularization

We can think of regularization as a way to make memorizing the data harder. As a result, the model is more prepared to handle different scenarios in the future. It is a bit like going running with a snorkel or swimming with a buoy in tow: by learning to overcome these conditions in training, we are better prepared for the real thing.

The simplest way to use a bias term is to add some random noise to the data. Because the error is random, the model can never fit it perfectly, so we help avoid overfitting. In social science, the bias term is often called an "**error term**" since we assume some measurement is associated with our inputs.

error term

However, randomness is hard for computers, so we often use normally (in the sense of Gaussian) distributed noise, which in itself is regular. It is already a strong assumption that the noise we encounter is normally distributed. In language, the error distribution might actually follow a power-law distribution.

Instead of guessing at the error distribution, we can tie the bias term to the parameters' distribution. Ideally, we would like a model that considers all features, rather than putting all its eggs in one basket (as in the "Tuesday" example above).

Essentially, such a model would distribute the weights for each feature relatively evenly. We can measure this by looking at the **L2 norm** of the weight vector: the root of the sum of squares of all weights. The more evenly distributed the weights are, the smaller this term becomes. The effect is that the model is forced to "hedge its bets" by considering *all* features, rather than putting all its faith in a few of them. It is, therefore, better equipped to deal with future cases. This process is called **L2 regularization**. The regularization term is usually weighted by a constant called λ.

L2 norm

L2 regularization

$$y = W \cdot \mathbf{X} + \lambda ||W||_2$$

The L2 norm uses the root of the sum of squares as an error term. Taking the square removes any signs on the elements:

$$||W||_2 = \sqrt{\sum_{i=1}^{N} w_i^2}$$

The L2 norm has unique analytical solutions, but it produces non-sparse coefficients (every element has a non-zero value). Therefore, it does not perform variable selection.

Ridge regression

When we use L2 as regularization for Logistic Regression, it is also known as **Ridge regression**.

L1 regularization

Theoretically, we can also minimize the L1 norm of the vector (**L1 regularization**). Here, we use the regular norm of the coefficient vector as the error term:

$$||W||_1 = \sum_{i=1}^{N} |w_i|$$

The L1 norm produces sparse coefficients, i.e., it sets many coefficients to 0, and therefore has many possible solutions for a given vector. Due to the vectors' sparsity, they all amount to an implicit variable selection: we can ignore all entries with a value of 0 and focus on the others. Using L1 regularization for Logistic regression is also known as **Lasso regression**.

Lasso regression

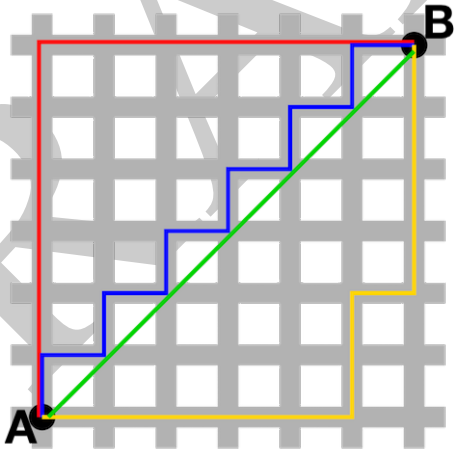


FIGURE 6. Examples of L1 vs. L2 norm in Euclidean space (modified from Wikipedia)

To see the difference between the two norms, see Figure 6: the red, blue, and yellow lines are all possible L1 solutions with the same value (12), whereas the green line is the L2 solution (8.49). Norms are closely related to distance measures like the Manhattan distance (equivalent to the L1 norm) or the Euclidean distance (which is equivalent to the L2 norm).

m

More rarely, we might use the **L0 norm** for regularization, which is the sum of all elements in the weight vector raised to their 0th power (i.e., either 0 or 1). That norm is simply the number of non-zero coefficients. It's a very aggressive way of eliminating uninformative features.

In practice, L2 regularization works best for prediction, though. In `sklearn`, logistic regression is already set to use L2 regularization, with a `C` parameter that stands for the λ weight. L1 is most useful in feature selection (see page 133). If we want to use L1, we have to explicitly select it via the `penalty` keyword:

```
lasso = LogisticRegression(penalty='l1')
```

CODE 58. Defining a Logistic Regression classifier with L1 regularization (Lasso).

8. Model Selection and Other Classifiers

There are, of course, plenty of other classifiers than Logistic Regression, and `sklearn` provides many of them. Thankfully, they all follow the same logic, so we can use the same `fit()` and `predict()` functions from before. In practice, we only need to change one line: where we define the classifier.

Note that each model has a specific set of parameters that govern its behavior (as the regularization parameter for Logistic Regression). To get the best classifier, we need to tune those parameters. This step is called **model selection**. A model is just one particular combination of parameters. To get the best possible result on our held-out data, we select the combination (model) that does that.

model selection

We can simply iterate over a set of possible values and record which one results in the highest performance:

```
from sklearn.metrics import cross_val_score

best_c = None
best_f1_score = 0.0

for c in [50, 20, 10, 1.0, 0.5, 0.1, 0.05, 0.01]:
    clf = LogisticRegression(C=c, n_jobs=-1)
    cv_reg = cross_val_score(clf, X, y=y, cv=5, n_jobs=-1, scoring="
    f1_micro").mean()

    print("5-CV on train at C={}: {}".format(c, cv_reg.mean()))
    print()

    if cv_reg > best_f1_score:
        best_f1_score = cv_reg
        best_c = c
```

```
16
17 print("best C parameter: {}".format(best_c))
```

CODE 59. Finding the best regularization parameter for Logistic Regression.

If you have few data points (less than 2000), trying a Support Vector Machine can help. If you have a lot of data points, Naive Bayes can be an option.

8.1. Support Vector Machines. A popular classifier is the Support Vector Machine (SVM). SVMs use a **kernel function** to project the data into a higher-dimensional space and take the similarities between instances in that space into account, which accounts for much of their predictive power. They also try to find the decision boundary between our classes that gives us the most "room" between them. It does that by computing **support vectors**, i.e., vectors parallel to the decision boundary (see Figure 7).

kernel function

support vectors

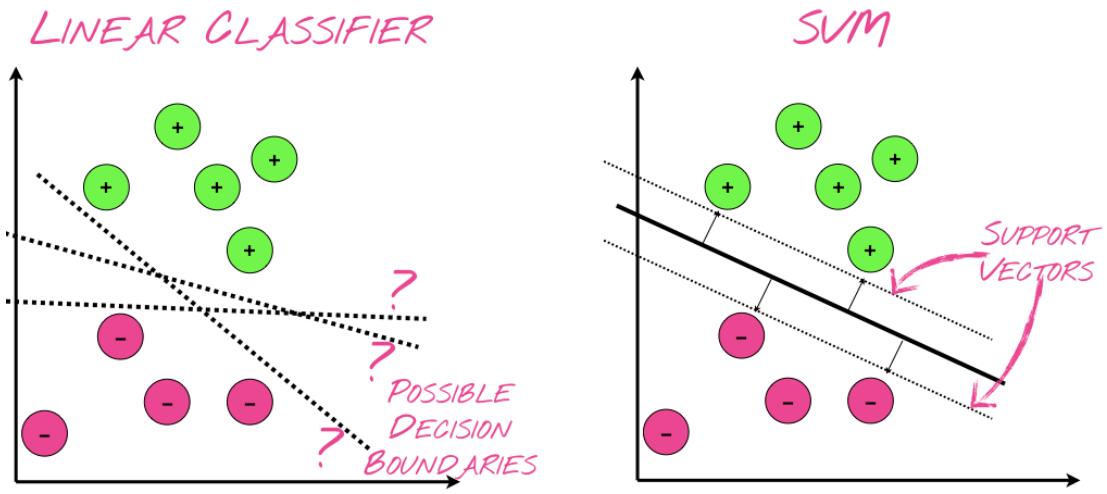


FIGURE 7. Comparison of a regular linear classifier and an SVM

SVMs, therefore, often have better performance than Logistic Regression on small to medium-sized data sets. However, they become too slow for more massive data sets. Performance depends crucially on the chosen kernel, which also requires optimization of various parameters. In essence, a kernel function implements a simple way to compute the similarity between two examples in a higher-dimensional space. More dimensions make it easier to find a **hyperplane** that separates our classes clearly, even if they were not separable in the original feature space. Some issues remain, though, see Figure 8.

hyperplane

In `sklearn`, SVM is implemented as `SVC` (the C stands for *classification*).

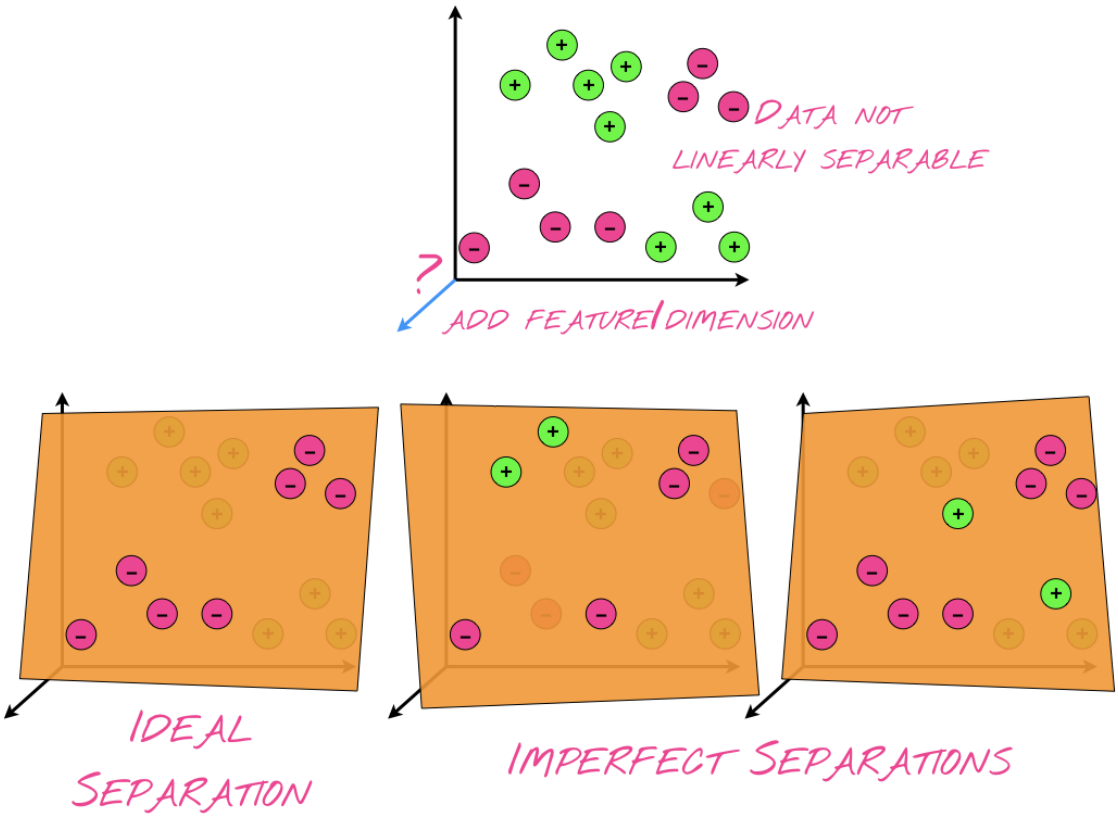


FIGURE 8. Example of better class separation by adding more dimensions.

```
from sklearn.svm import SVC
svm = SVC()
```

CODE 60. The SVM classifier in `sklearn`.

The **time complexity** of an algorithm (e.g., how many operations we need to fit a model to the data) depends on the input. Specifically, to train a classifier, it depends on the number of observations and features. For Logistic Regression, both of those are roughly linear, so if we double the number of observations and features, it will take twice as long to fit the data. For an SVM, these numbers are cubic for the observations and quadratic for the features. So doubling the observations and features will take the algorithm exponentially longer to fit. This increase is due to the objective function of the SVM, which involves both computing a kernel matrix (i.e., the similarity of all instances with each other) and solving a quadratic function.

time complexity

The difference is mainly due to two facts: 1) SVMs can not be parallelized, Logistic Regression can be. The first fact is partially due to 2) the SVM has a more complex objective function: rather than finding *a* decision boundary, the SVM tries to find the

support vectors
slack

decision boundary that maximizes the distance between classes (the **support vectors** that give the classifier its name), while allowing for some **slack** (i.e., observations being on the wrong side of the decision boundary).

For details on the time complexity of classification algorithms, see Mohri et al. (2018).

8.2. Naive Bayes. The Naive Bayes (NB) algorithm is a very simple generative model. The generative story (i.e., how the observed feature vector was generated) behind NB is simple: we pick a class according to some distribution $P(\mathbf{y})$. Based on that label, we then produce each of the observed feature values with some probability $P(\mathbf{X}|y)$. The features are independent of one another, i.e., feature 2 is not influenced by features 1 or 3 (or any other, for that matter), only by the label. This assumption is, of course, somewhat simplistic (hence the name *Naive* Bayes), especially when our features are words in a sentence. Words tend to occur together following specific patterns, not independently of each other. In practice, though, NB works reasonably well, and it has three advantages. 1) it is easy to implement, 2) it is very straightforward to interpret the model, 3) it scales really well to enormous amounts of data because 4) it parallelizes well.

All we have to do is go through our instances, count how often we see each label, and normalize by the total number of instances. That gives us the distribution over class labels $P(y)$. Similarly, we count how often each feature occurs with each class and divide that by the number of times we have seen that class. That gives us the conditional probability distribution $P(X|y)$ of features given a class. To label a new instance, we simply sum all our conditional probabilities from the present features, compute all class probabilities, and then pick the highest one as a label. Or, mathematically:

$$\hat{y} = \arg\max_{y_i \in \mathbf{y}} P(y_i) \prod_{j=1}^{N} P(x_j|y_i)$$

Here, $\arg\max$ just means "pick the value that gives you the highest result for the following equation".

In `sklearn`, NB is implemented in various forms, but the one we would use with discrete features is the multinomial NB:

```
from sklearn.naive_bayes import MultinomialNB
nb = MultinomialNB()
```

CODE 61. NB in `sklearn`.

fractional or expected counts

NB can also be made to incorporate unlabeled data. If we have no label, we need to compute how often we would see each class *hypothetically*. This quantity is called **fractional or expected counts**. We can then use those counts as a new distribution to compute how often we would have seen the features activated. This part is the E-step of the EM algorithm. In the end, we normalize our fractional counts to make them probability distributions again. These should explain the data a bit better, i.e., when we

compute the likelihood that these parameters generated the data ($P(y)$ and $P(X|y)$), it should be higher than when we started.

We can repeat the E and M-steps indefinitely. We will always get a little bit better in terms of data likelihood, but at some point, it will not make a difference anymore. To measure that, we can compute the difference between the data likelihood of the previous iteration and the current one. When that difference is smaller than some threshold, we stop. Alternatively, we can run for a fixed number of iterations and stop after that. In practice, people typically set a maximum number of iterations but also measure the data likelihood.

9. Model Bias

Models are their own source of biased predictions. Concretely, the tendency of a model to rely on minor differences between subjects to satisfy the objective function and make reasonable predictions (**overamplification**). Unchecked, the model can maximize its objective score by amplifying the difference when predicting new data and creating a larger imbalance than in the original data. Essentially, the model might give the correct answers, but for the wrong reasons. Yatskar et al. (2016) have shown that in an image captioning data set, 58% of the captions for pictures of a person in a kitchen mentioned women. A small but noticeable difference, which could be corrected by sampling. However, as Zhao et al. (2017) showed, a standard statistical model trained on this slightly biased data ended up predicting the gender of a person in a kitchen picture to be a woman in 63% of the cases.

overamplification

The cost of these false positives seems low. A user might be puzzled or amused when seeing a mislabeled image or receiving an email addressing them with the wrong gender. However, relying on models that produce false positives may lead to **bias confirmation** and **overgeneralization**. Some of these false positives might amuse us, but would we accept them if the system predicted sexual orientation and religious views rather than age and gender? For any text prediction task, this is just a matter of changing the target variable and finding some data.

bias confirmation
overgeneralization

Another problem of overamplification is the proliferation of stereotypes. Rudinger et al. (2018) found that coreference resolution systems (which link a pronoun to the noun they refer to) were biased by gender. In the sentence, “The surgeon could not operate on her patient: it was her son,” the model does not link “surgeon” and “her.” The cause of this inability is presumably biased training data, but the effect feeds into gender stereotypes. Similarly, Kiritchenko and Mohammad (2018) showed that sentiment analysis models changed their scores for the same sentences when they replaced a single word. E.g., replacing a female with a male pronoun or replacing a typically “white” name with a typically “black" name. Both “She/He made me feel afraid” and “I made Heather/Latisha feel angry" result in higher scores for the second case.

9.0.1. *Counter measures.* To address the overgeneralization of models, we can ask ourselves, “would a false answer be worse than no answer?” Instead of taking a *tertium non datur* approach to classification, where a model has to (and will) produce *some*

answer, we can use dummy variables that say "unknown." We can also use measures such as error weighting, which incur a higher penalty if the model makes mistakes on the smaller class. Alternatively, we can use confidence thresholds below which we do not assign a label.

adversarial learning

Recently, (Li et al., 2018) have shown that **adversarial learning** (a specialized architecture in neural networks) can reduce the effect of predictive biases. It even helps to improve the performance of the models!

10. Feature Selection

We have our data, we have decided on a classifier, and made sure it does not overfit. However, we are not sure we maxed out the performance with these choices. Can we do more to improve performance?

If we are using a BOW representation and a linear model, we can fit a classifier (with regularization!), and then inspect the magnitude of the coefficients. This **feature selection** can assist us in improving performance by choosing a smaller, more informative set of features. Naturally, if we use feature selection to improve performance, we have to do it on the training data. We cannot peak at the development or the test data!

feature selection

However, analyzing features can give us an exploratory overview of the correlation between our features and the target class. This correlation can be informative to discover new facts. For example, which words best describe each political candidate in a primary (see `https://www.washingtonpost.com/news/monkey-cage/wp/2016/02/24/these-6-charts-show-how-much-sexism-hillary-clinton-faces-on-twitter`).

10.1. Dimensionality Reduction. We have looked at dimensionality reduction earlier in the context of visualization (see page 94). However, we can also use it to improve our performance. Rather than fitting a model on a large set of features, we can use dimensionality reduction techniques. They allow us to find a smaller number of dimensions that still capture the variation in the data, but provide fewer chances to overfit by exploiting spurious patterns in the data. This lower-dimensional representation could, for example, be the $\mathbf{U}$ matrix from SVD.

If we use a lower-dimensional version of a BOW matrix, the individual dimensions do not mean anything anymore. I.e., the columns of our matrix are no longer counts or TFIDF values. We cannot map them back to the terms we used as features.

10.2. Chi-Squared. χ^2 (or Chi-squared) is a test that measures the correlation between a positive feature and the categorical outcome variable.

In `sklearn`, we can use the `feature_selection` package to implement the selection process with χ^2, by sorting the features according to their correlation, and keeping only the top k.

```
from sklearn.feature_selection import SelectKBest
from sklearn.feature_selection import chi2

selector = SelectKBest(chi2, k=1500).fit(X, y)
X_sel = selector.transform(X)
print(X_sel.shape)
```

CODE 62. Selecting the top 1500 features according to their χ^2 value.

10.3. Randomized Regression. Logistic regression is a generalized linear model, where the log odds of the response probability is a linear function of the independent variables. The coefficients of these variables are estimated via maximum likelihood, and are then subsequently interpreted. If we use the model on word count data and a target category, we can analyze the importance of individual terms by inspecting the magnitude of the coefficients for those terms.

However, this approach presupposes that we have already selected the independent variables. In the case of text, however, the vocabulary size, and therefore the number of independent variables, can be massive. Yet their individual occurrences are sparse (some words occur only once even in large corpora). It is often impossible to know a priori which variables to select.

There are several ways in which we can potentially select a subset of informative variables: We can remove variables with non-significant co-occurrence statistics. However, this requires a large data set and does not guarantee the best model. We can fit all possible models on the entire data and pick the one with the best fit, using some measure of fitness. However, the large number of possible models makes this approach computationally prohibitively extensive. It is also not robust to overfitting. We can fit a model on the entire data, and use an L1 penalty term. This constraint causes uninformative variables to receive 0 weight during fitting. However, this method is highly reliant on extensive data and sufficient observations. To address the last concern, we can randomize the previous method. We collect repeated random subsets of the data, fit an L1-penalized (Lasso) model to each and aggregate the coefficient vectors. Variables that frequently receive a high coefficient are then selected. When the target variable is binary, this process is randomized logistic regression (or Stability Selection, see Meinshausen and Bühlmann (2010))

Intuitively, we simulate a range of different conditions, and pick the variables that are informative independent of the condition. In practice, we fit 200-1000 independent Logistic Regression models on the data, each on a random subset sampled with replacement. For each model, we use a different weight for the L1 regularization parameter, controlling how aggressively coefficients are driven to 0. Variables that have a score of at least 0.5 (i.e., they received a positive coefficient weight in over 50% of the models) can be considered informative.

```
from sklearn.linear_model import LogisticRegression
import numpy as np

def positive_indicatorsRLR(X, y, target, vectorizer,
    selection_threshold=0.3, num_iters=100):
    n_instances, n_feats = X.shape

    pos_scores = [] # all coefficient > 0
    neg_scores = [] # all coefficient < 0
    # choices for lambda weight
```

```
    penalities = [10,5,2,1,0.5,0.1,0.05,0.01,0.005,0.001,0.0001,
    0.00001]

    # select repeated subsamples
    for iteration in range(num_iters):
        # initialize a model with randomly-weighted L1 penalty
        clf = LogisticRegression(penalty='l1', C=penalities[np.random
    .randint(len(penalities))])

    # choose a random subset of indices of the data with replacement
        selection = np.random.choice(n_instances, size=int(
    n_instances * 0.75))
        try:
            clf.fit(X[selection], y[selection])
        except ValueError:
            continue

    # record which coefficients got a positive or negative score
        pos_scores.append(clf.coef_ > 0)
        neg_scores.append(clf.coef_ < 0)

    # normalize the counts
    pos_scores = (np.array(pos_scores).sum(axis=0)/num_iters).reshape
    (-1)
    neg_scores = (np.array(neg_scores).sum(axis=0)/num_iters).reshape
    (-1)

    # find the features corresponding to the non-zero coefficients
    features = vectorizer.get_feature_names()
    pos_positions = [i for i, v in enumerate(pos_scores >=
    selection_threshold) if v]
    neg_positions = [i for i, v in enumerate(neg_scores >=
    selection_threshold) if v]

    pos = [(features[i], pos_scores[i]) for i in pos_positions]
    neg = [(features[i], neg_scores[i]) for i in neg_positions]

    posdf = pd.DataFrame(pos, columns='term score'.split()).
    sort_values('score', ascending=False)
    negdf = pd.DataFrame(neg, columns='term score'.split()).
    sort_values('score', ascending=False)

    return posdf, negdf
```

CODE 63. Randomized Regression for variable selection.

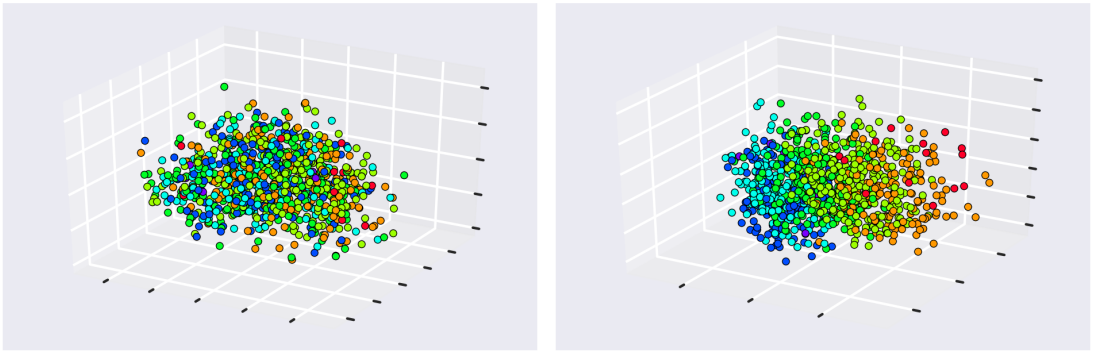


FIGURE 9. Effect of retrofitting on 500 instances in a ten-class embedding space. Each color represents one class label. Before retrofitting (left), instances are based on linguistic information. After retrofitting on labels (right), classes are more clearly separated.

11. Retrofitting to Increase Within-Class Similarity

We have previously seen (page 94) how we can use retrofitting to include non-linguistic information in embeddings. However, the same technique can be used to make classes more linearly separable (Hovy and Fornaciari, 2018). Figure 9 shows an example for 500 data points from ten classes (each class is labeled with a different color). Retrofitting sorts the instances into clusters based on color.

Instead of external information, we can also use the class labels as equivalence classes, i.e., $\Omega = \{(d_i, d_j | y_i = y_j)\}$. In essence, we separate classes in the embedding space, making them easier to differentiate by a classifier. We achieve this separation by defining the neighborhood of each instance to be all instances with the same class label. This neighborhood definition effectively improves classification performance.

However, we can only retrofit the embeddings of authors in the training set D_{train}, since we need information about the class label in order to construct Ω. However, the retrofitting process changes the configuration of the embedding space (into $\hat{D}_{train}$), so a separating hyperplane learned on $\hat{D}_{train}$ will not be applicable to a test set D_{test} in the original embedding space.

In order to extend the homophily information to instances in the test set, we use a *translation matrix* T (a 300×300 matrix), which approximates the transformation from the original training data matrix D_{train} into the retrofitted matrix $\hat{D}_{train}$. We obtain T by minimizing the least-square difference in $D_{train} \cdot T = \hat{D}_{train}$.

T captures the retrofitting operation, and allows us to modify the test instances' representations *as if* their classes were known, despite the absence of label information. In particular, by applying T to the embeddings in the unlabeled test set D_{test}, we obtain a retrofitted version $\hat{D_{test}}$ that preserves the transformation learned on the training data. Since the least-square approximation is not perfect, we find that in practice fitting a

classifier on the approximation $D_{train} \cdot T$ works better than using $\hat{D}_{train}$, acting as a regularizer.

12. Structured Prediction

In all of the previous chapters, we have looked at documents as individual elements. A document was either a word on its own or a sentence as the sum of its words. In classification, we try to find the best single label for this input.

However, one of the core properties of language is that it is sequential. We perceive words to follow each other, and to have a position in a sentence. What if we want to find a sequence of labels, one for each word in the sentence? Say we have a sentence like "They can can cans", and we want to find the parts of speech (noun, verb, adjective, etc.) for each word. This task is called **part-of-speech tagging**, and is one of the early success stories of NLP.

part-of-speech tagging
POS tagging

We could try to just generate all possible tag sequence for the sentence (see Figure 10) and use each of those unique sequences as a possible label in a classification task. But there is a problem...

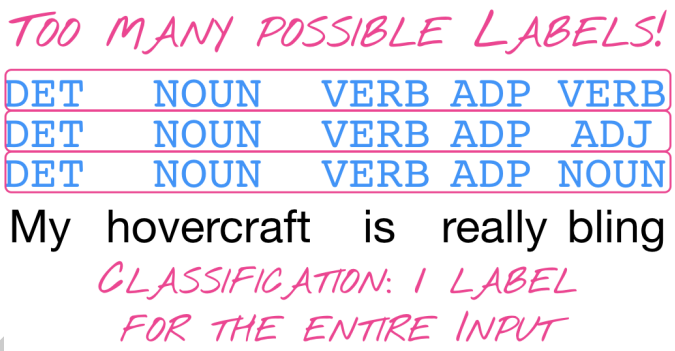


FIGURE 10. Naive listing of all possible tag sequences

Say each word has on average 1.2 tags, and a sentence has 17 words, like this one. That's 1.2^{17}, or about 22 possible labels. For just one short sentence! Some sentences have 40 or more words, meaning 1470 or more unique labels. And we have to add up all those unique labels... Clearly, we cannot afford to do that, or we would end up with a $10,000$-way classification. We will have to do something else.

Sequence is also important in grammatical and semantic long-range dependencies. Nouns can occur both before and after verbs. They act either as the subject of a sentence, or as its object. I.e., either as the thing doing the action (the verb), or as the thing something is done to. For example, the subject-verb agreement (here, *er* and *begegnet*) in this German sentence:

> *"Wenn er aber auf der Strasse der in Sammt und Seide gehüllten jetzt sehr ungenirt nach der neusten Mode gekleideten Regierungsräthin begegnet."*[6]

Such long-range dependencies are also important for relation extraction. Say we wanted to extract the fact `founded_by(Amazon, Jeff Bezos)` from the sentence "Jeff Bezos, or what Dr. Evil would look like on steroids, went from book seller to billionaire after he founded Amazon in 1994." The relevant parts are far apart, and we would have to completely ignore the inserted clause. Dependency is also important to determine the **scope** of the negation in "This is *not* in any sense of the word a *funny* movie."

scope

All of these examples are cases for **structured prediction**. In structured prediction, we want to find the best sequence of labels, given a sequence of words. The sequential nature of sentences makes words **context-dependent**: they can mean very different things, depending on the context in which they occur. They are **ambiguous**. In the sentence "I will see the show tonight", the word "show" is a noun. However, in the sentence "Let me show you something", the same word is a verb. So to decide the best label for each word, we have to take the context into account.

structured prediction

context-dependent

ambiguous

There are some regularities that help us. Examples of "show" as a noun include sentences like "His show was cancelled" or "We went to see a show together." Other examples of "show" as a verb are "She wanted to show that she could do it" or "They show us a good time."

Even though the words are all different, there are some regularities in the tags of the preceding words. In both cases of "show" as a noun, the preceding word is a determiner. In both cases of the verb "to show", the preceding word is a pronoun.

So to determine the part of speech of a specific word, we need to look at the part of speech of the preceding words. Of course, their part of speech, in turn, depends on their preceding words' parts of speech, etc. This is starting to look like infinite recursion.

There are, however, two facts that can help us break that recursion cycle. The first helper are unambiguous words. Many words have only one possible tag. Not only does that make prediction extremely trivial, it also helps us to find regularities in tag sequences. The second helping fact is that there is a natural end to the infinite recursion: if we go back far enough, we reach the first word of a sentence. And the tag history of all first words is the same: we can imagine that the sentence beginning has its own unambiguous special tag, say `START`.

These two facts together give us a good starting point to tackle the structured prediction problem. Guess the tag for the first word, given that the previous tag was START, then proceed in the same manner for each word (see Figure 11).

[6]The example is from Mark Twain's essay *The Awful German Language*, and it's slightly old-fashioned, but only slightly exaggerated.

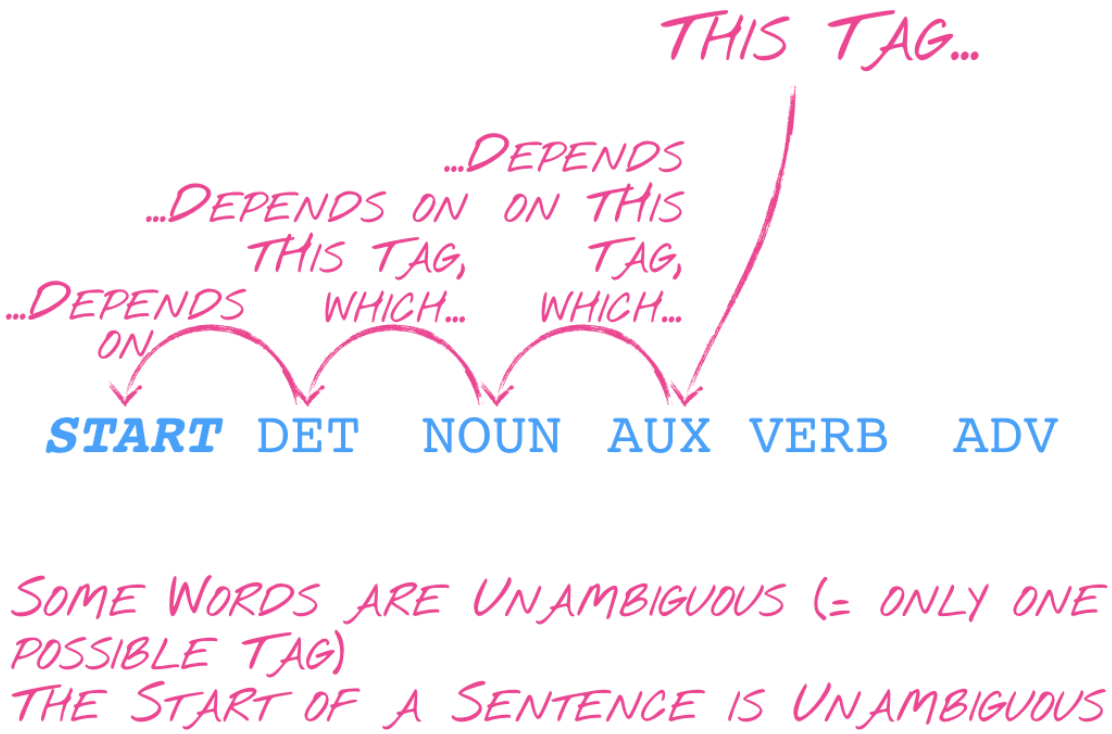


FIGURE 11. Imagining a START tag for the sentence beginning helps.

12.1. The Structured Perceptron. The algorithm that implements this intuition is the **Structured Perceptron** Collins (2002). The central prediction problem it solves can be expressed mathematically as

Structured Perceptron

$$T|W = \arg\max_{Y \in \mathcal{Y}} \sum_{d=1}^{D} \theta_d \cdot \Phi_d(W, Y)$$

In other words, we are given a sequence of words, W (a vector of N words, $w_1, w_2, ..., w_N$). If we are searching the best tag sequence T (which is a vector of tags $t_1, t_2, ..., t_N$), we return the best label sequence Y (out of all possible label sequences $\mathcal{Y}$). To get Y, we compute the sum of all features Φ multiplied with their respective weight θ.

Each of those features is the result of a function that examines W and the predicted tags Y.

$$\Phi_d(W, T) = \sum_{i=1}^{N} \phi_d(w_i, t_{i-2}, t_{i-1}, W, i)$$

The global features of a sentence, Φ are simply the sum of all the local features, ϕ_i, derived from the words $w_1, w_2, ..., w_N$. Each local feature is derived from the word w_i, the two previous (true or predicted) tags t_{i-2}, t_{i-1}, the sentence as a whole (W), plus an indicator of the position, i.

To set the weights, we update each θ_d after each sentence, moving the weight closer to the features derived from the true label sequence and further away from the predicted ones:

$$\theta_d = \theta_d + \Phi_d(W, T) - \Phi_d(W, Y)$$

If the features on the predicted tags are the same as the features of the true tags, we don't need to update.

While the math can look intimidating, the algorithm itself is relatively simple. In pseudo-code, it only takes a few lines:

```
for each iteration:
    for words, true_tags in examples:
        features = get_features(words, true_tags)
        prediction = predict(features)
        if prediction != true_tags:
            for feature in features:
                weights[feature][true_tags] += 1
                weights[feature][prediction] -= 1
```

CODE 64. Pseudo-code for the Structured Perceptron.

The Structure: Hidden Markov Models. The assumed structure underlying the algorithm is a second order **Hidden Markov Model**. HMMs are used in a variety of applications, but they follow a very simple idea. What we can see (the words) was generated by something we cannot see (the tag sequence), one step at a time. I.e., someone first created a sequence of tags, and then turned each of them into a word. This is our **generative story**. It sounds a bit strange (this is not how we usually form sentences), but it captures the notion that many sentences have the same syntactic structure.

Hidden Markov Model *HMM*

generative story

And that helps us model the effect of previous tags. Each tag is influenced by the ones that came before it. This assumption is one of the so-called **Markov properties**. To make things easier, we assume that each state depends only on some of the previous steps, not all previous states. In a second-order HMM, each tag depends on the two previous ones (see Figure 12). This history dependence is the **Markov Chain** part of the HMMs. What makes it a *hidden* Markov model is the assumption that these tags are unobservable.

Markov properties

Markov Chain

You might wonder why we need to make all of these assumptions? Why don't we just take the best tag for each word and be done? Remember that words are ambiguous: the most likely tag for "can" is one of NOUN, VERB, or AUX. But what is the most likely tag in "the can"? Probably no longer VERB or AUX... The context disambiguates it. That's why we want the Markov Chain of hidden tags.

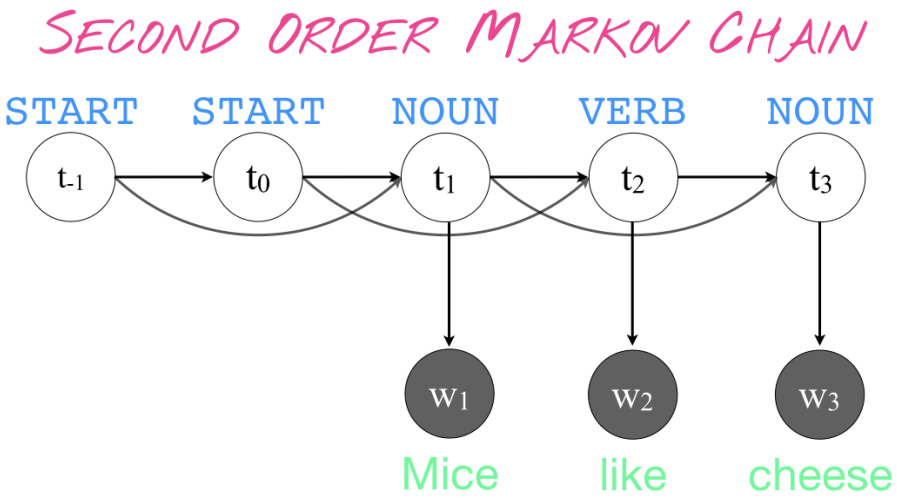


FIGURE 12. A second-order Hidden Markov Model

The tags are hidden, but they are connected to something we can observe, namely the words. And that connection follows certain regularities we can capture with features: words ending in "–ing" are more likely to be nouns or verbs, words starting with an uppercase letter are probably proper names, etc. (Note that we also assume that previous words do not influence the current word. This is another Markov property, the independence of the observations from one another).

Assuming an underlying structure like an HMM lets us reverse-engineer that generative story, which in turn gives us the most likely sequence of tags. That is what the inference in the Structured Perceptron uncovers. We reward good features that result in the right tags by increasing their weight, and we decrease the weight of bad features.

12.1.1. *Inference.* We can choose between two inference algorithms to get the prediction we need to update our weights. The first algorithm is **greedy inference**. It is fast, but gives only approximate solutions. In practice, those solutions are pretty good, though. Greedy inference chooses the highest scoring tag as prediction at each step, then moves on. It can therefore be tripped up in longer dependencies.

greedy inference

Viterbi decoding

The second inference algorithm, **Viterbi decoding**, does take context explicitly into account. It chooses each tag based on its score and the score of its best predecessor (which in turn is computed by the same approach). Viterbi decoding is exact, and can

work better than greedy inference if we have little data. However, it is more involved, and therefore slower.

For either inference, we make use of unambiguous words to speed up the process and to avoid errors that could affect accuracy further on.

Greedy Inference. In greedy inference, we look at each word in turn. We derive its features from the tag history, context, and word itself, and use them to compute the scores for each possible tag. We pick the highest-scoring tag for this word as output, and then we move on. The tag we just predicted now becomes part of the tag history for the next word, and on we go. Because it is a series of simple decisions, we can move through it quickly. All we have to do is iterate over the words. The pseudocode is short and simple:

```
for each words, tags:
    predictions = []
    for each word in words:
        scores = get_features(word, tags[-2], tags[-1], words) *
    weights
        tag = argmax(scores)
        predictions.append(tag)
```

CODE 65. Pseudo code for greedy inference decoding.

If we made a wrong decision early in the sentence, it sticks with us. In fact, it becomes the basis of future predictions, which could be affected by this decision. For example, in the sentence "Attack was their only option," the word "attack" could be labeled as VERB, which would change the scores for subsequent predictions. In practice, though, greedy inference is fairly robust. It was used in the early taggers of `spacy`, which was famous for its speed and accuracy.

Viterbi Decoding. Viterbi inference is slightly more involved than greedy inference. For each word, we compute a joint score that consists of the regular feature score and the score we computed for the best previous tag. We always carry this history around, and only if we find the best score for the current word do we decide on the best tag for the previous word. That lag allows us to fix mistakes. It does, however, require some overhead in bookkeeping, and a few extra loops.

Since we essentially compute a path through the sentence, going from tag to tag, we use a structure called a lattice or trellis: an interwoven band of paths through the sentence. To keep track of the score and the best predecessor tag, we use two matrices with the same dimensions. Both have one row for each tag in our tag set, and one column for each token in our sentence.

The scoring matrix is often called Q. $Q[i, j]$ denotes the score of the best paths leading us up to node word j with tag i. Since we do not need to find a best predecessor for the first word (it's START), the first column of Q is always just the scores of the different tags given the START history. In each subsequent column, each cell is the

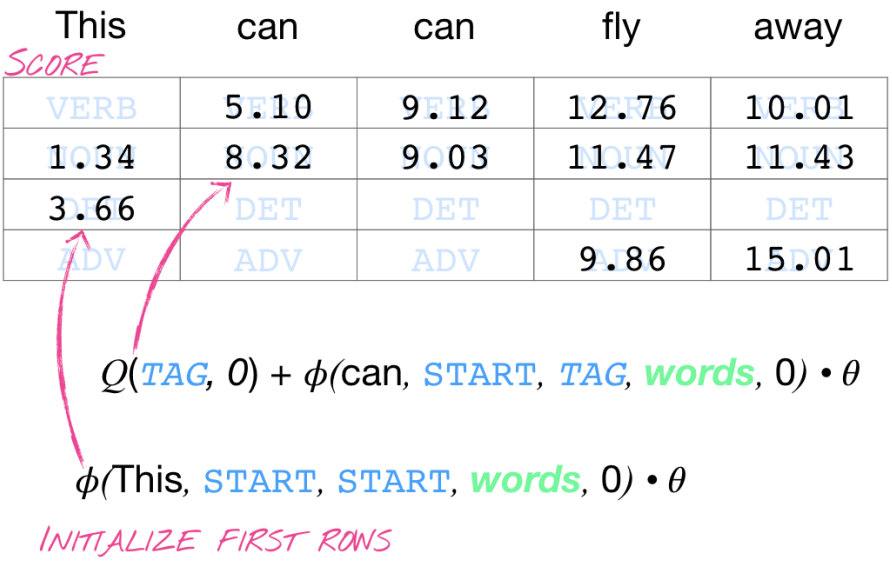


FIGURE 13. Initialization of Q matrix in Viterbi decoding.

sum of all best paths arriving there, i.e., each tag score multiplied by the Q of the best predecessor (see Figure 13).

The backpointer matrix (or lattice) has the same shape, but it is literally a lookup table. The value k in the cell at position i, j tells us that if we tag word j with tag i, the best predecessor tag is k (for word $j - 1$).

We first walk through the sentence, filling the Q and backpointer matrices. For each position, we need to iterate over all possible tags of the previous two positions, to compute the combined score of the current tag and the best predecessor. This is where the tag dictionary for words comes in handy. We only have to iterate over all tags when we get an unknown word.

Once we have reached the end of the sentence, we find the highest scoring tag for the last token, and add it to the output. We then look up the corresponding best predecessor to that tag, and from there simply follow the backpointers, adding each to the output (see Figure 14). Since we walk through the sentence backwards, we need to reverse the result before returning it.

```
# initialize scores
features = get_features(words[0], START, START, context, 1)
scores = get_scores(features)

for each allowed tag on word 1:
    Q[tag, word] = scores[tag]
```

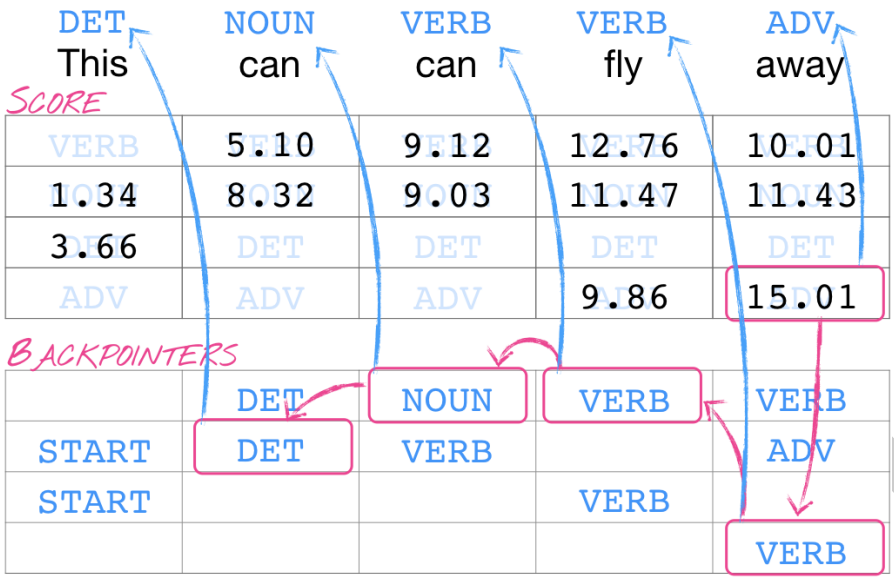


FIGURE 14. Decoding via backpointers in Viterbi inference.

```
7
8  # fill lattice for every position and tag with Viterbi score Q
9  for each word i:
10     for each allowed prev_tag on prev_word:
11         best_score = float('-Inf')
12         # score of previous tag on previous word
13         prev_score = Q[prev_tag, prev_word]
14
15         for each allowed prev2_tag:
16             if i == 1:
17                 prev2_tag = START
18
19             # get features of word i with the two previous tags
20             features = get_features(word, prev2_tag, prev_tag,
       context, i)
21             scores = get_scores(features)
22
23             # update best score
24             for each allowed tag on current word:
25                 tag_score = prev_score + scores[tag]
26
27                 if tag_score > best_score:
```

```
                    Q[tag, word] = tag_score
                    best_score = tag_score
                    backpointers[tag, word] = prev_tag

# find best tag for last word
best_tag = argmax(Q[:, -1])
prediction = [best_tag]

for i in range(N-1,0,-1):
    next = backpointers[best_tag, i]
    prediction.append(next)
    best_tag = next

return reversed(prediction)
```

CODE 66. Pseudo code for Viterbi decoding.

As you can see, the pseudo code is more involved, and has more levels of loops. However, the last line in the main loop, where we set the backpointers, is what makes this inference powerful: we decide on the best tag for a word only after we have seen the next one.

The entire code for the Structured Perceptron adds a few convenience functions for saving and loading of trained models, some bookkeeping structures, and output.

The code here follows the paper by Collins (2002), with some updates for the greedy inference from the Explosion blog posy by Matthew Honnibal (`https://explosion.ai/blog/part-of-speech-pos-tagger-in-python`).

```
class StructuredPerceptron(object):
    """
    implements a structured perceptron as described in Collins 2002,
    with updates from https://explosion.ai/blog/part-of-speech-pos-
    tagger-in-python
    """

    def __init__(self):
        """
        initialize model parameters
        """
        self.tags = set()
        self.feature_weights = defaultdict(lambda: defaultdict(float)
    ) #feature_name -> tags -> weight
        self.weight_totals = defaultdict(lambda: defaultdict(float))
    #feature_name -> tags -> weight
        self.timestamps = defaultdict(lambda: defaultdict(float)) #
    feature_name -> tags -> weight
```

```
15
16          self.tag_dict = defaultdict(set) #word -> {tags}
17
18          self.START = "__START__"
19          self.END = "__END__"
20
21
22      def normalize(self, word):
23          """
24          lowercase word, and replace numbers, user names, and URLs
25          """
26          return re.sub(urls, 'URL', re.sub(users, '@USER', re.sub(
    numbers, '0', word.strip().lower())))
27
28
29      def evaluate(self, data_instances, predict_method='greedy'):
30          correct = 0
31          total = 0
32          for (words, tags) in data_instances:
33              preds = self.predict(words, method=predict_method)
34              matches = sum(map(lambda x: int(x[0]==x[1]), zip(preds,
    tags)))
35              correct += matches
36              total += len(tags)
37          return correct/total
38
39
40      def fit(self, file_name, dev_file=None, iterations=10,
    learning_rate=0.25, inference='greedy', verbose=False):
41          """
42          read in a CoNLL-format file, extract features to train weight
     vector
43          """
44          # initialize tag dictionary for each word and get tag set
45          instances = [(words, tags) for (words, tags) in self.
    read_conll_file(file_name)]
46          for (words, tags) in instances:
47              self.tags.update(set(tags))
48
49              for word, tag in zip(words, tags):
50                  self.tag_dict[self.normalize(word)].add(tag)
51
52          if dev_file:
53              dev_instances = [(words, tags) for (words, tags) in self.
    read_conll_file(dev_file)]
```

```
54
55          # iterate over data
56          for iteration in range(1, iterations+1):
57              correct = 0
58              total = 0
59              if verbose:
60                  print('Iteration {}'.format(iteration+1), file=sys.
    stderr, flush=True)
61                  print("*" * 15, file=sys.stderr, flush=True)
62
63              random.shuffle(instances)
64              for i, (words, tags) in enumerate(instances):
65                  if i > 0:
66                      if i%1000==0:
67                          print('%s'%i, file=sys.stderr, flush=True)
68                      elif i%20==0:
69                          print('.', file=sys.stderr, flush=True, end='
    ')
70
71                  # get prediction
72                  prediction = self.predict(words, method=inference)
73
74                  # derive global features
75                  global_gold_features, global_prediction_features =
    self.get_global_features(words, prediction, tags)
76
77                  # update weight vector:
78                  # 1. move closer to true tag
79                  for tag, fids in global_gold_features.items():
80                      for fid, count in fids.items():
81                          nr_iters_at_this_weight = iteration - self.
    timestamps[fid][tag]
82                          self.weight_totals[fid][tag] +=
    nr_iters_at_this_weight * self.feature_weights[fid][tag]
83                          self.timestamps[fid][tag] = iteration
84                          self.feature_weights[fid][tag] +=
    learning_rate * count
85
86                  # 2. move further from wrong tag
87                  for tag, fids in global_prediction_features.items():
88                      for fid, count in fids.items():
89                          nr_iters_at_this_weight = iteration - self.
    timestamps[fid][tag]
90                          self.weight_totals[fid][tag] +=
    nr_iters_at_this_weight * self.feature_weights[fid][tag]
```

```
91                          self.timestamps[fid][tag] = iteration
92                          self.feature_weights[fid][tag] -=
    learning_rate * count
93
94                  # compute training accuracy for this iteration
95                  correct += sum([int(predicted_tag == true_tag) for
    predicted_tag, true_tag in zip(prediction, tags)])
96                  total += len(tags)
97
98                  # output examples
99                  if verbose and i%1000==0:
100                     print("current word accuracy:{:.2f}".format(
    correct/total))
101                     print(list(zip(words,
102                                    [self.normalize(word) for word in
    words],
103                                    tags,
104                                    prediction)), file=sys.stderr,
    flush=True)
105
106             print('\t{} features'.format(len(self.feature_weights)),
    file=sys.stderr, flush=True)
107             print('\tTraining accuracy: {:.2f}\n'.format(correct/
    total), file=sys.stderr, flush=True)
108             if dev_file:
109                 print('\tDevelopment accuracy: {:.2f}\n'.format(self.
    evaluate(dev_instances, method=inference)), file=sys.stderr, flush
    =True)
110
111         # average weights
112         for feature, tags in self.feature_weights.items():
113             for tag in tags:
114                 total = self.weight_totals[feature][tag]
115                 total += (iterations - self.timestamps[feature][tag])
     * self.feature_weights[feature][tag]
116                 averaged = round(total / float(iterations), 3)
117                 self.feature_weights[feature][tag] = averaged
118
119
120     def get_features(self, word, previous_tag2, previous_tag, words,
    i):
121         """
122         get all features that can be derived from the word and
    previous tags
123         """
```

```
124         prefix = word[:3]
125         suffix = word[-3:]
126 
127         features = {
128                     'PREFIX={}'.format(prefix),
129                     'SUFFIX={}'.format(suffix),
130                     'LEN<=3={}'.format(len(word)<=3),
131                     'FIRST_LETTER={}'.format(word[0]),
132                     'WORD={}'.format(word),
133                     'NORM_WORD={}'.format(words[i]),
134                     'PREV_WORD={}'.format(words[i-1]),
135                     'PREV_WORD_PREFIX={}'.format(words[i-1][:3]),
136                     'PREV_WORD_SUFFIX={}'.format(words[i-1][-3:]),
137                     'PREV_WORD+WORD={}+{}'.format(words[i-1], words[i
    ]),
138                     'NEXT_WORD={}'.format(words[i+1]),
139                     'NEXT_WORD_PREFIX={}'.format(words[i+1][:3]),
140                     'NEXT_WORD_SUFFIX={}'.format(words[i+1][-3:]),
141                     'WORD+NEXT_WORD={}'.format(word, words[i+1]),
142                     'NEXT_2WORDS={}+{}'.format(words[i+1], words[i
    +2]),
143                     'PREV_TAG={}'.format(previous_tag),
        # previous tag
144                     'PREV_TAG2={}'.format(previous_tag2),
         # two-previous tag
145                     'PREV_TAG_BIGRAM={}+{}'.format(previous_tag2,
    previous_tag),  # tag bigram
146                     'PREV_TAG+WORD={}+{}'.format(previous_tag, word),
                  # word-tag combination
147                     'PREV_TAG+PREFIX={}_{}'.format(previous_tag,
    prefix),         # prefix and tag
148                     'PREV_TAG+SUFFIX={}_{}'.format(previous_tag,
    suffix),         # suffix and tag
149                     'WORD+TAG_BIGRAM={}+{}+{}'.format(word,
    previous_tag2, previous_tag),
150                     'SUFFIX+2TAGS={}+{}+{}'.format(suffix,
    previous_tag2, previous_tag),
151                     'PREFIX+2TAGS={}+{}+{}'.format(prefix,
    previous_tag2, previous_tag),
152                     'BIAS'
153                 }
154         return features
155 
156 
157     def get_global_features(self, words, predicted_tags, true_tags):
```

```
158         '''
159         sum up local features
160         '''
161         context = [self.START] + [self.normalize(word) for word in
    words] + [self.END, self.END]
162 
163         global_gold_features = defaultdict(lambda: Counter())
164         global_prediction_features = defaultdict(lambda: Counter())
165 
166         prev_predicted_tag = self.START
167         prev_predicted_tag2 = self.START
168 
169         for j, (word, predicted_tag, true_tag) in enumerate(zip(words
    , predicted_tags, true_tags)):
170             # get the predicted features. NB: use j+1, since context
    is longer than words
171             prediction_features = self.get_features(word,
    prev_predicted_tag2, prev_predicted_tag, context, j+1)
172 
173             # update feature correlation with true and predicted tag
174             global_prediction_features[predicted_tag].update(
    prediction_features)
175             global_gold_features[true_tag].update(prediction_features
    )
176 
177             prev_predicted_tag2 = prev_predicted_tag
178             prev_predicted_tag = predicted_tag
179 
180         return global_gold_features, global_prediction_features
181 
182 
183     def get_scores(self, features):
184         """
185         predict scores for each tag given features
186         """
187         scores = defaultdict(float)
188 
189         # add up the scores for each tag
190         for feature in features:
191             if feature not in self.feature_weights:
192                 continue
193             weights = self.feature_weights[feature]
194             for tag, weight in weights.items():
195                 scores[tag] += weight
196 
```

```
197         # return tag scores
198         if not scores:
199             # if there are no scores (e.g., first iteration),
200             # simply return the first tag with score 1
201             scores[list(self.tags)[0]] = 1
202 
203         return scores
204 
205 
206     def predict(self, words, method='greedy'):
207         '''
208         predict tags using one of two methods
209         '''
210         if method == 'greedy':
211             return self.predict_greedy(words)
212         elif method == 'viterbi':
213             return self.predict_viterbi(words)
214 
215 
216     def predict_viterbi(self, words):
217         '''
218         predict using Viterbi decoding
219         '''
220         context = [self.START] + [self.normalize(word) for word in
    words] + [self.END, self.END]
221 
222         N = len(words)
223         M = len(self.tags) #number of tags
224         tags = sorted(self.tags)
225 
226         # create trellis of size M (number of tags) x N (sentence
    length)
227         Q = np.ones((M, N)) * float('-Inf')
228         backpointers = np.ones((M, N), dtype=np.int16) * -1 #
    backpointers
229 
230         # initialize probs for tags j at position 1 (first word)
231         features = self.get_features(words[0], self.START, self.START
    , context, 1)
232         scores = self.get_scores(features)
233         allowed_initial_tags = self.tag_dict[context[1]]
234 
235         for j in range(M):
236             if not allowed_initial_tags or tags[j] in
    allowed_initial_tags:
```

```
237                 Q[j,0] = scores[tags[j]]
238 
239         # filling the lattice, for every position and every tag find
    viterbi score Q
240         for i in range(1, N):
241             allowed_tags = self.tag_dict[context[i+1]]
242 
243             # for every previous tag
244             for j in range(M):
245                 best_score = 0.0#float('-Inf')
246                 prev_tag = tags[j]
247 
248                 # skip impossible tags
249                 allowed_previous_tags = self.tag_dict[context[i]]
250                 if allowed_previous_tags and prev_tag not in
    allowed_previous_tags:
251                     continue
252 
253                 best_before = Q[j,i-1] # score of previous tag
254 
255                 # for every possible pre-previous tag
256                 for k in range(M):
257                     if i == 1:
258                         prev2_tag = self.START
259                     else:
260                         prev2_tag = tags[k]
261                         # skip impossible tags
262                         allowed_previous2_tags = self.tag_dict[
    context[i-1]]
263                         if allowed_previous2_tags and prev2_tag not
    in allowed_previous2_tags:
264                             continue
265 
266                     # get features of word i with the two previous
    tags
267                     features = self.get_features(words[i], prev2_tag,
     prev_tag, context, i+1)
268                     scores = self.get_scores(features)
269 
270                     # update best score
271                     for t in range(M):
272                         tag = tags[t]
273                         # if word is unknown, use all tags, otherwise
     allowed ones
274                         if not allowed_tags or tag in allowed_tags:
```

```
                                tag_score = best_before + scores[tag]

                                if tag_score > best_score:
                                    Q[t,i] = tag_score
                                    best_score = tag_score
                                    backpointers[t,i] = j

        # final best
        best_id = Q[:,-1].argmax()

        # print best tags in reverse order
        predtags = [tags[best_id]]

        for i in range(N-1,0,-1):
            idx = backpointers[best_id, i]
            predtags.append(tags[idx])
            best_id = idx

        #return reversed predtags
        return predtags[::-1]


    def predict_greedy(self, words):
        '''
        greedy prediction
        '''
        context = [self.START] + [self.normalize(word) for word in
    words] + [self.END, self.END]

        prev_predicted_tag = self.START
        prev_predicted_tag2 = self.START

        out = []

        for j, word in enumerate(words):
            # for unambiguous words, just look up the tag
            predicted_tag = list(self.tag_dict[context[j+1]])[0] if
    len(self.tag_dict[context[j+1]]) == 1 else None

            if not predicted_tag:
                # get the predicted features. NB: use j+1, since
    context is longer than words
                prediction_features = self.get_features(word,
    prev_predicted_tag2, prev_predicted_tag, context, j+1)
                scores = self.get_scores(prediction_features)
```

```
316
317                 # predict the current tag
318                 predicted_tag = max(scores, key=scores.get)
319
320             prev_predicted_tag2 = prev_predicted_tag
321             prev_predicted_tag = predicted_tag
322
323             out.append(predicted_tag)
324
325         return out
326
327
328     def read_conll_file(self, file_name):
329         """
330         read in a file with CoNLL format:
331         word1    tag1
332         word2    tag2
333         ...      ...
334         wordN    tagN
335
336         Sentences MUST be separated by newlines!
337         """
338         current_words = []
339         current_tags = []
340
341         with open(file_name, encoding='utf-8') as conll:
342             for line in conll:
343                 line = line.strip()
344
345                 if line:
346                     word, tag = line.split('\t')
347                     current_words.append(word)
348                     current_tags.append(tag)
349
350                 else:
351                     yield (current_words, current_tags)
352                     current_words = []
353                     current_tags = []
354
355         # if file does not end in newline (it should...),
356         # check whether there is an instance in the buffer
357         if current_tags != []:
358             yield (current_words, current_tags)
359
360
```

```
361     def save(self, file_name):
362         """
363         save model as pickle file
364         """
365         print("saving model...", end=' ', file=sys.stderr)
366         with open(file_name, "wb") as model:
367             # pickle cannot save default_dictionaries
368             # => make copy and turn into regular dictionaries
369             save_feature_weights = defaultdict(lambda: defaultdict(
    float))
370             save_feature_weights.update(self.feature_weights)
371             save_tag_dict = defaultdict(set)
372             save_tag_dict.update(self.tag_dict)
373 
374             save_feature_weights.default_factory = None
375             save_tag_dict.default_factory = None
376             pickle.dump((save_feature_weights, save_tag_dict, self.
    tags),
377                         model, -1)
378         print("done", file=sys.stderr)
379 
380 
381     def load(self, file_name):
382         """
383         load model from pickle file
384         """
385         print("loading model...", end=' ', file=sys.stderr)
386         with open(file_name, 'rb') as model:
387             try:
388                 parameters = pickle.load(model)
389             except IOError:
390                 msg = ("No such model file.")
391                 raise MissingCorpusError(msg)
392 
393             feature_weights, tag_dict, tags = parameters
394             self.tags = tags
395 
396             # pickle cannot store defaultdicts, so we need a 2-step
    process
397             # 1. initialize
398             self.feature_weights = defaultdict(lambda: defaultdict(
    float))
399             self.tag_dict = defaultdict(set)
400 
401             # 2. update
```

```
402             self.feature_weights.update(feature_weights)
403             self.tag_dict.update(tag_dict)
404         print("done", file=sys.stderr)
405         return None
```

CODE 67. Code for a Structured Perceptron class.

Two functions to point out are `get_features()` and `get_scores()`. The first derives all features from a word, its tag history, and the general context. This includes all possible combinations and analysis, and can be extended. Nothing in this code is specific to POS tagging, though it might be necessary to add other features for other tasks. To allow both greedy and Viterbi inference, we use a helper function that sums up all word features to a global feature vector for the sentence.

The scoring function takes the features as input. For each feature, it iterates over all tags, and looks up the weights associated with this feature for the current tag. Summing up the scores for all features for each tag give us the ranking of tags.

The feature weights are averaged, i.e., weighted by their respective frequency. This requires some additional bookkeeping, but substantially helps performance.

Note that the weights are updated after each sentence, weighted by the learning rate. Because we update so frequently, the weights can be pushed in a certain direction if we see the same features several times in a row. To break this symmetry, we shuffle the order of instances after each iteration.

You might wonder why to bother with this model, given that there are very powerful neural networks for sequence (which we will cover in the next chapters). The Structured Perceptron might not be as powerful as, say an LSTM, but it has two things going for it. It needs much less data to achieve decent performance, and it is interpretable. We can look at the feature weights and find exactly what features influence the decision.

13. Neural Networks Background

Imagine you are trying to predict a complex phenomenon, like human behavior. Simple, linear models can help explain what is happening by creating a simpler version of the problem. However, they will not be able to do the complex relationship between inputs and outputs justice. To better approximate the complexity of the problem, we need a more complex model. Non-linear models can match the complexity of the problem, and are thus much more adept at analyzing them. However, we buy this complexity with a decrease in explainability.

So far, we have relied on simple, linear, feature-based models for prediction. These models perform reasonably well, and they have the added benefit that they can be interpreted. However, linear models can fail if the underlying distribution is inherently non-linear—and in many language-related studies, exactly this is the case. It is therefore not too surprising that neural models have recently revolutionized language-related tasks, and went from academic designs to production-strength systems. It started with speech processing, reaching almost human voice recognition and speech-to-text capabilities, and continued with a quantum leap in the quality of machine translations. The effect of networks on NLP has variously been compared to a car on a "rabbit in the headlights",[7] or as a "tsunami" (Manning, 2015). Despite this violent imagery, the net effect on the field has been positive, as it provided new avenues in prediction and generation (e.g., in chatbots, for picture descriptions, etc.).

neural networks

feature engineering

In this chapter, we will look at the currently dominant class of non-linear models, **neural networks**. They have opened up new ways to do research. They have excellent predictive performance, and they can learn representations, which has freed us from cumbersome **feature engineering**. Instead, they use word embeddings, which allowed us to spend more time experimenting with model architectures that best capture the underlying problem.

Neural networks are a very active, fast-developing research area. There are new, fundamental developments almost at a monthly level. In 2018, a debate about whether one model type or another was better for text classification went on via conference papers over several months. Recommendations and best practices changed at the same speed. There are plenty more open issues out there, and they grow even more numerous by the day. In order to truly get into the topic, we would therefore need much more space than we have here, as the mathematical underpinnings alone require a substantial treatment of various linear algebra concepts. And even then, it would be largely outdated by the time this appeared. Instead, we will here mainly look at the *intuition* behind neural networks, and look at a few select model architectures to give you an idea of the range.

[7] https://inverseprobability.com/2016/05/15/nlp-is-a-rabbit-in-the-headlights

For a more in-depth, excellent introduction to the use of neural networks in NLP, see Goldberg (2017), or the shorter primer on which it is based (Goldberg, 2016). For an intuitive explanation and great visualization, see the excellent videos by 3Blue1Brown `https://www.youtube.com/watch?v=aircAruvnKk&list=PLZHQObOWTQDNU6R1_67000Dx_ZCJB-3pi`. For more implementation details in `keras`, see Chollet (2017).

13.1. Neural History. While neural networks have dominated machine learning news over the last few years, in reality, they are an old technology. The first version, called the **Perceptron**, which was inspired by actual neurons, was introduced in the 1950s by Rosenblatt (1958). For an overview of applications in political science, see Chatsiou and Mikhaylov (2020).

Perceptron

After the great initial excitement, there were some setbacks. It became apparent that the Perceptron itself was limited (Minsky and Papert, 1969, proved that perceptrons could not learn the exclusive OR logic function, see below). However, it eventually became clear that this model was a great building block for more complex systems. We can build **layers** by running several simple perceptrons over the same input and using their output as input for another perceptron. The resulting model is a **multilayer perceptron** or **feed-forward neural network** (these terms are used interchangeably, but they mean the same thing). The more layers (i.e., the deeper the stack of layers, hence deep learning), the more expressive the model becomes, and the more complex phenomena it can tackle.

layers

multilayer perceptron

feed-forward neural network

Many fields have now seen revolutionary breakthroughs using these neural networks. In NLP, the progress has been led by using word embeddings, and then expanded to the very flexible model architectures. We have seen a lot of innovation, and many tasks have significantly improved. The most publicly visible example of this shift is probably the translation quality of services like Google Translate (Wu et al., 2016).

The one drawback neural networks have for social scientists is their lack of interpretability. As great as they are for correctly predicting an outcome, it is tough to explain why they do so. This problem has the same origin as the networks' strength: their distributed nature. Each level in the architecture specializes during training, and by taking different paths from input to output, we can model theoretically any relationship between them. Computer vision research (an early adopter of neural networks) has shown that different layers in the network correspond to different visual correlates. E.g., the first layers recognize lines, then simple shapes, then parts of faces, and ultimately things like generic faces for facial recognition. Unfortunately, for language, matters are less straightforward. The initial hope with networks was that they would mirror the brain, and that the layer would correspond to different levels of linguistic analysis, i.e., phonology, morphology, syntax, and semantics (Dell, 1986). So far, this has not panned out. Using an architectural feature called **attention** (Bahdanau et al., 2014), we can now at least show the influence individual words or passages have. (It also turns out to be very useful for performance). However, the use of neural networks for explanation is

attention

still minimal. However, neural networks are still a very fast developing research area, and new algorithms come out constantly.

To better understand the power and potential of networks, we will first review the basics, and then introduce elements of networks that have proven useful.

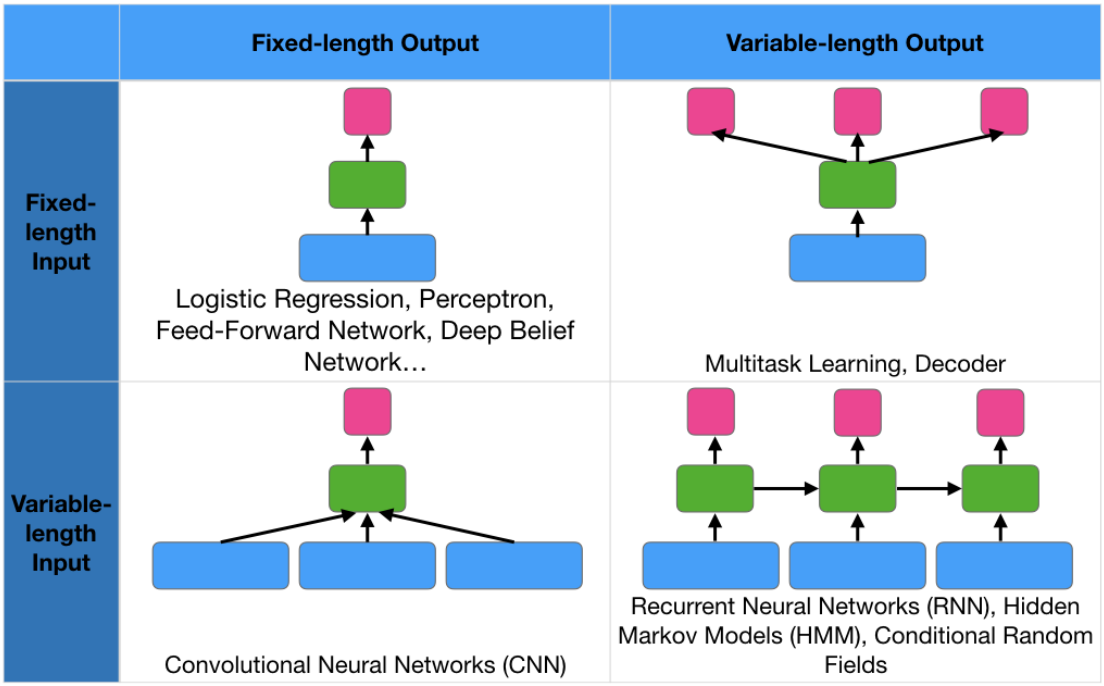


FIGURE 15. The different types of neural architectures.

13.2. Network Basics. Generally, neural networks for prediction can be distinguished into four classes, based on the type of input and output (see also Figure 15):

perceptron

(1) fixed-length input, fixed-length output: e.g., classify a word embedding as dialect or standard. This uses a feed-forward neural architectures like the **perceptron** (see 13.3).

convolutional neural network

(2) variable-length input, fixed-length output: e.g., classify a text (i.e., a sequence of word embeddings) as positive, negative, or neutral. This is the use case for **convolutional neural network**s (see 14.1).

recurrent neural network

(3) variable-length input, variable-length output: e.g., labeling each word with its part of speech, or translating a German text into a French text. This uses **recurrent neural network**s (see 14.2).

(4) fixed-length input, variable-length output: e.g., text generation based on a prompt (this uses a decoder architecture), or the classification of multiple properties of a document embedding (this uses a **multitask learning** architecture). **multitask learning**

13.2.1. *Training.* The basic idea behind training a network is that of show and tell. The network is initialized randomly and then presented with input-output pairs. We take the input, run it through the network, and see whether it produces the desired output. The most useful explanation of this process comes from a New York Times article on AI (Lewis-Kraus, 2016):

> *"Imagine that you're playing with a child. You tell the child, 'Pick up the green ball and put it into Box A.' The child picks up a green ball and puts it into Box B. You say, 'Try again to put the green ball in Box A.' The child tries Box A. Bravo."*

This example describes a model that only has an input and an output layer. However, as we add more layers to get more power, things get a lot more complicated, especially when the prediction does not match the output. From the same article (emphasis my own):

> *"Now imagine you tell the child, 'Pick up a green ball, go through the door marked 3 and put the green ball into Box A.' The child takes a **red** ball, goes through the **door marked 2** and puts the red ball into **Box B**. How do you begin to correct the child? You cannot just repeat your initial instructions, because the child does not know at which point he went wrong."*

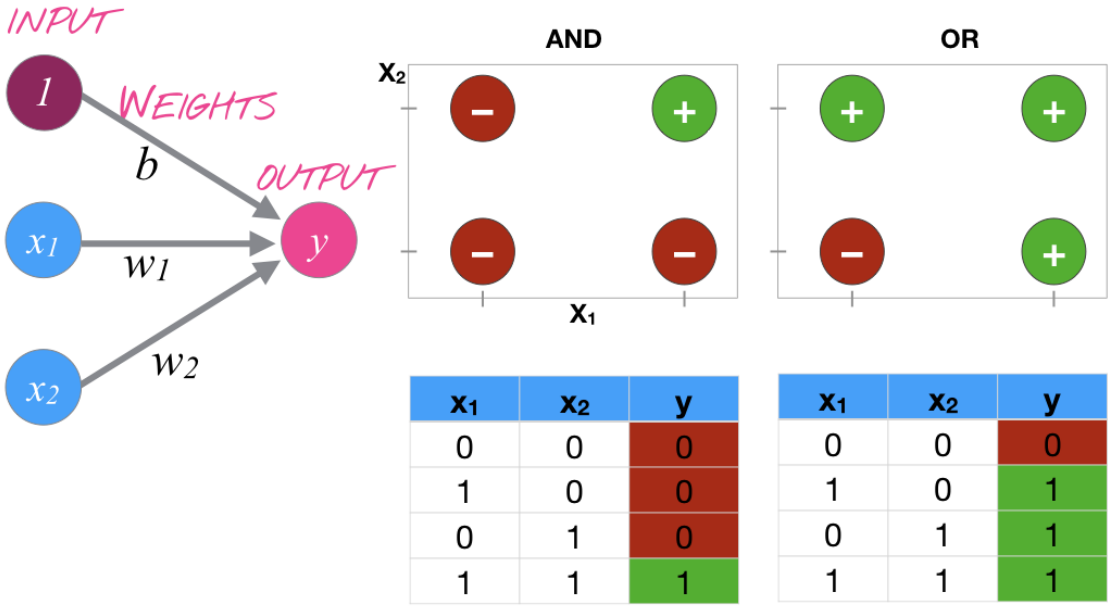
If the prediction does not match the output, we use an **error function** to compute how *different* the prediction was from the intended output. We then walk back from the output through each layer of the network, adjusting the parameters at each step in such a way that they produce the correct output the next time around. This step is called **backpropagation**, and it is what sets the parameters of the model. Obviously, the updates we need to make at each layer depend on the updates we needed to make at each previous layer. **error function** **backpropagation**

Normally, we stop the backpropagation at the last layer before we reach the input. If instead, we include the input in the updating process, we are essentially also learning the best way to represent the problem for the task, i.e., through embeddings. This part is therefore called **representation learning** (if we started with random embeddings) or **finetuning** (if we use existing embeddings). **representation learning** **finetuning**

During backpropagation, we have to compute at each layer (e.g., ball color, door, box) what went wrong. Then we can tell the previous layer what parameters it has to correct in order to give the current layer the right input. What are those model parameters we optimize in backpropagation? As we will see they are simply vectors and matrices of weights.

13.3. The Perceptron. The grandfather and basic building block of any neural network is the perceptron (Rosenblatt, 1958). The Perceptron was inspired by the observation that actual neurons fire when they receive a signal that exceeds a certain threshold. We can also imagine this (maybe less grandiose, but more apt) like a smoke detector. It has an array of sensors (say, one on each side, or maybe at different heights), it sums up the readings from all of them. If the total amount of smoke measured exceeds a certain limit, it rings the alarm. The Perceptron works similarly: it weighs the information from the different inputs, and if their total information crosses a threshold, it produces an output.

$$f(X) = a(w_1 x_1 + w_2 x_2 + b) \qquad \hat{y} = \begin{cases} +1 & \text{if } f(X) \geq 0 \\ -1 & \text{otherwise} \end{cases}$$



x1	x2	y
0	0	0
1	0	0
0	1	0
1	1	1

x1	x2	y
0	0	0
1	0	1
0	1	1
1	1	1

FIGURE 16. Visualizations, logical tables, architecture, and equations for AND and OR perceptrons.

Rosenblatt (1958) showed that a perceptron with two binary inputs and one binary output could learn to solve simple logic functions like AND and OR (see Figure 16 for a visualization and the logical tables of both) with the same basic architecture. The input value for each node is simply 0 or 1, and the output of the Perceptron is $+1$ or -1.

To represent a logical AND, the perceptron checks whether the sum of the weighted inputs to 2. If they do, that means both input nodes were on (or true), and therefore the output is true, and the Perceptron outputs $+1$.

To represent a logical OR, it is enough to check whether the sum of the inputs is equal to 1 (i.e., if at least one of the inputs was true). If they do, then the output is true. You can see the architecture and equations for both perceptrons in Figure 16. However, so far, we have to explicitly specify the threshold, depending on whether we wanted to recognize AND or OR functions. It turns out that we can let the Perceptron learn that threshold, so that we can use the same Perceptron for both AND and OR functions. To do that, we first add another input node that is always on (which is called the **bias term**). The bias node has its own bias weight, which corresponds to the threshold needed for the respective functions.

bias term

The summed value from all the inputs is now a linear combination of the two input values, each multiplied with their respective weight, plus the bias term (which is the input value 1 multiplied by its bias weight). We can write that as a function over the inputs X, which is a vector with the input values, x_1 and x_2:

$$f(X) = w_1 x_1 + w_2 x_2 + b$$

However, we still need to test whether $f(X)$ exceeds a certain threshold. Since we could have weights of any size, $f(X)$ could have almost any value. To get it back into a defined range, we use a special function that limits the possible range to a minimum and maximum. No matter how large or small $f(X)$ is, it can never exceed that range. I.e., we squeeze the result of $f(X)$ to be within a desired range. This squeezing results in an S-like curve if we plot all values of $f(X)$ and their squeezed value. There are different functions available that achieve this S shape, and they are jointly referred to as **sigmoid functions**. Some of them are shown in Figure 17.

sigmoid functions

Depending on which sigmoid function we choose, the range can be from -1 to 1, or from 0 to 1. Once we have such a defined range, our threshold simply becomes the midpoint between the minimum and maximum value. For example, if we use the hyperbolic tangent ($tanh$) function, which squeezes everything into a range from -1 to 1, our threshold becomes 0. The original Perceptron used a logistic function as sigmoid, which ranges from 0 to 1.

Once we have that threshold value, anything above it is the positive output class, everything below the negative output class. Our prediction $\hat{y}$ is $+1$ if we are above the threshold, and -1 if we are below it. Applying the sigmoid transformation of $f(x)$ and testing whether it exceeds the threshold for the perceptron to fire is called an **activation function**.

activation function

This architecture was now able to learn *both* logical AND and OR functions, simply by learning the appropriate threshold for each from the data. People were excited!

It got even more exciting when it turned out that the Perceptron can do more than just learn AND and OR functions. Its weights essentially define a decision boundary between positive and negative examples. In essence, we can use the Perceptron to

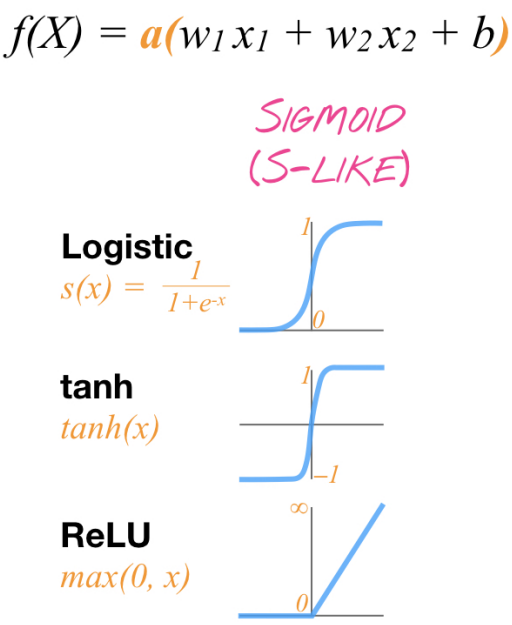


FIGURE 17. Activation functions and their output

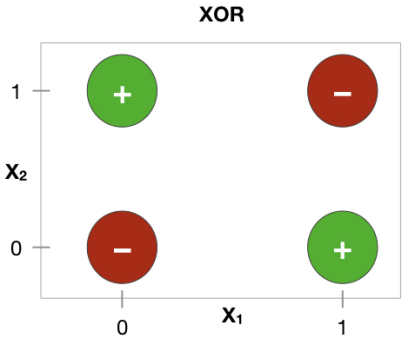
predict the labels of new data!

The problem was that this simple architecture broke down when faced with the *exclusive or* (XOR) problem: the output is only true if one input is true and the other one false. If both are true or both are false, then the output is false. No matter how you choose the threshold, you can never produce the right output for all inputs. This is an example of **linearly non-separable** problem. We cannot draw a decision boundary between the examples of different classes. This result from (Minsky and Papert, 1969) dealt the Perceptron and the development of AI a major blow. Excitement abated, and the development of AI stalled for a while.

linearly non-separable

Marsland (2011) has a friendly introduction to the Perceptron and neural networks in Python.

```
1 import numpy as np
2
3 def sigmoid(x):
4     '''
5     ranges from  0 to 1
6     '''
7     return 1 / (1 + np.exp(-x))
8
```



x_1	x_2	y
0	0	0
1	0	1
0	1	1
1	1	0

FIGURE 18. Visualization and logical table for the XOR function.

```
class Perceptron:
    def __init__(self, num_inputs):
        # initialize the weights randomly, and set the bias to 1
        self.w1 = np.random.random(num_inputs)
        self.b1 = 1

    def predict(self, X):
        # compute activation for input layer
        fX = np.dot(X, self.w1) + self.b1
        # non-linear transform
        activation = sigmoid(fX)
        # check threshold: for sigmoid, use 0.5, for tanh, use 0
        y = np.where(fX >= 0.5, 1, -1)
        return y

    def fit(self, train_data, train_labels, num_epochs=20):
        models = []
        print(num_epochs)
        for epoch in range(1, num_epochs+1):
            print(epoch)
            for (X, y) in zip(train_data, train_labels):
                pred_label = self.predict(X)
```

```
                if pred_label != y:
                    print('update')
                    self.w1 = self.w1 + (X * y)
                    self.b1 = self.b1 + y

            models.append((self.w1, self.b1))

        return models
```

CODE 68. A simple Perceptron implementation.

We can use this Perceptron to fit the basic logic functions it was originally developed to match:

```
and_data = np.array([[1, 1], [1, 0], [0, 1], [0, 0]])
and_labels = np.array([1, -1, -1, -1])

or_data = np.array([[1, 1], [1, 0], [0, 1], [0, 0]])
or_labels = np.array([1, 1, 1, -1])

xor_data = np.array([[1, 1], [1, 0], [0, 1], [0, 0]])
xor_labels = np.array([-1, 1, 1, -1])

# initialize perceptron
perceptron = Perceptron(2)

iters = perceptron.fit(and_data, and_labels, num_epochs=10)
and_predictions = perceptron.predict(and_data)
```

CODE 69. Fitting a Perceptron on simple logic functions.

Obviously, this version is somewhat overly simple. In order to have a more powerful and flexible version, we can use the `keras` library in Python

```
from keras.models import Model
from keras.layers import Input, Dense

# input: a sequence  of 2 integers
main_input = Input(shape=(2,), name='main_input')

# add the output layer
output = Dense(2, activation='hard_sigmoid', name='output',
    kernel_initializer='glorot_uniform')(main_input)

# f(X) = sigmoid(X*W + b)

```

```
# the model is specified by connecting input and output
perceptron_keras = Model(inputs=[main_input], outputs=[output])

# compile the model
perceptron_keras.compile(loss='binary_crossentropy',
              optimizer='sgd',
              metrics=['accuracy'])

# train the model on a validation set and dave the loss and accuracy
   for every epoch
history = perceptron_keras.fit(X, y,
                    epochs=15,
                    verbose=1,
                    validation_split=0.2
                )
```

CODE 70. Perceptron implementation in the `keras` functional API.

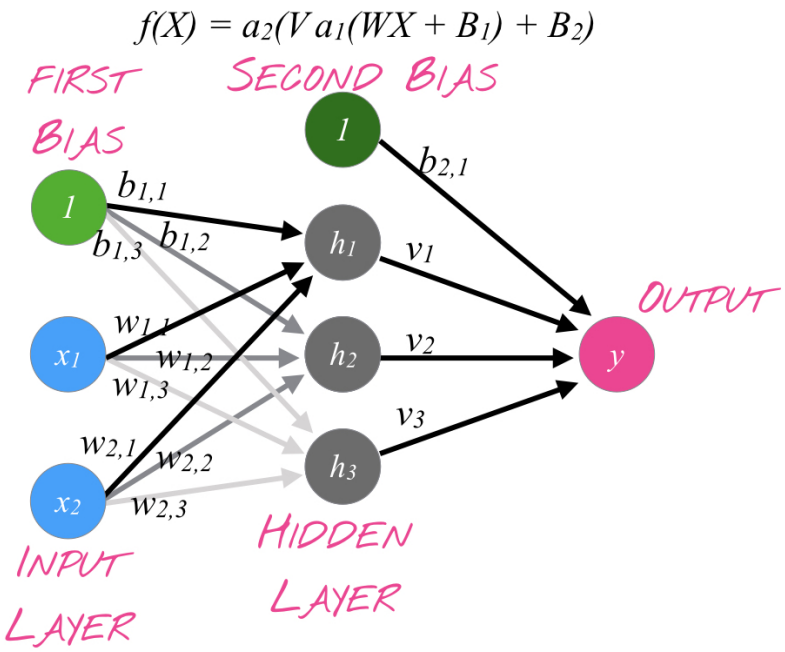


FIGURE 19. Architecture and equation for the multilayer perceptron

13.4. The Multilayer-Perceptron. We can graph the two inputs and the resulting output for the AND and OR problems, as in Figure 16. We can see that the two output classes can be separated by drawing a straight line (called the **decision boundary**). However, for the XOR problem, we can not draw any linear decision boundary that

decision boundary

separates the two output classes (see Figure 18). The operative word for the decision boundary in the XOR problem turns out to be *linear*. To overcome the XOR problem, the model needed to use a more flexible decision boundary. We can not use a straight line to separate the classes, but we can very well separate the classes if we use a curve. The problem was how to do this programmatically.

A curve, it turns out, is nothing more than a more complex function. So the solution to the XOR problem was simple. We use several perceptrons, all on the same input, and then add another perceptron on top, which uses the outputs of the first set of perceptrons as input. Because this input to the final Perceptron is not "visible" (i.e., it is not part of the problem we are given, but generated by us), it is called a **latent layer**.

latent layer

The resulting architecture is a lot more complex. It needs a lot more weights, including several bias weights (see Figure 19), but it is capable of solving the XOR problem. In fact, we can theoretically approximate *any* underlying distribution, no matter how complicated it is, by using several non-linear functions in a row. Because of this property, neural networks are **universal function approximators**, and they have done extremely well in cases where the input and output are indeed connected via a complicated function.

universal function approximators

This architecture has other advantages: If we want more than two output classes, we can simply add more (binary) output nodes. The predicted output is then a vector of output scores, and the prediction is simply the ID of the node with the highest activation score.

As you can see from the diagram in Figure 19, these models quickly become a confusing mess of nodes, weights, and their indices. To implement and compute all this efficiently, it is easier to use vectors and matrices. Figure 20 shows how the explicit notation with indices looks as matrices and in matrix notation. We have already discussed how to represent our input and output as these structures. By representing biases and weights as matrices as well, we can use operations like dot products to quickly calculate the activation, error, or prediction.

```
# input: a sequence  of 2 integers
mlp_input = Input(shape=(2,), name='main_input')

# add a hidden layer
mlp_hidden = Dense(16, activation='relu', name='hidden',
    kernel_initializer='glorot_uniform')(mlp_input)

# add the output layer
mlp_output = Dense(2, activation='softmax', name='output',
    kernel_initializer='glorot_uniform')(mlp_hidden)

# the model is specified by connecting input and output
mlp = Model(inputs=[mlp_input], outputs=[mlp_output])

```

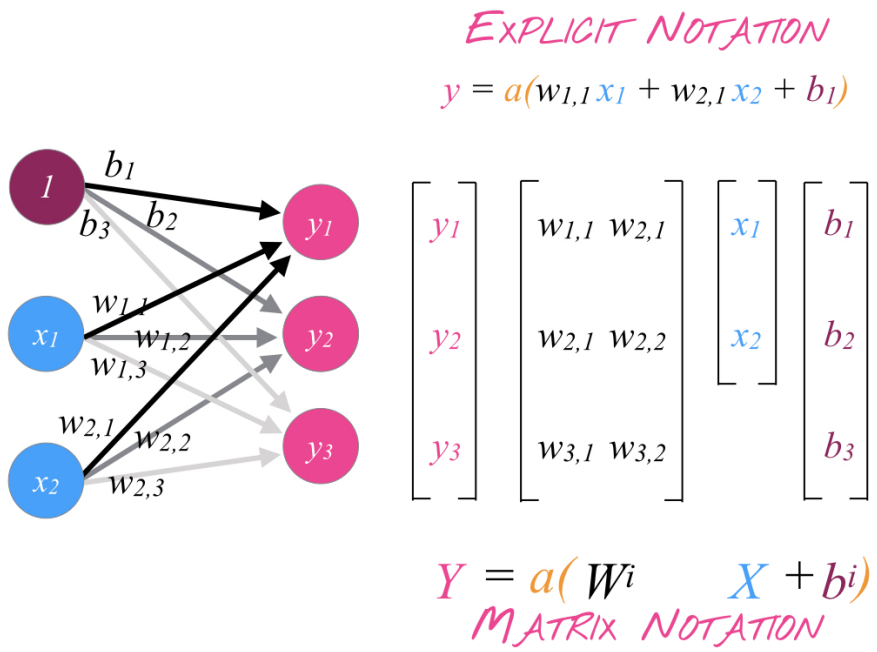


FIGURE 20. Visualization of matrix notation for multi-class perceptron

```
mlp.compile(loss='binary_crossentropy',
              optimizer='sgd',
              metrics=['accuracy']
            )

mlp_history = mlp.fit(X, y,
                      epochs=50,
                      verbose=1,
                      validation_split=0.2)
```

CODE 71. Multilayer Perceptron in `keras`.

13.5. Computation. In order to visualize (and compute) the prediction and updates of the training, it can be helpful to use a **computational graph**. Figure 21 shows the steps involved in computing the error for a prediction on the sentence "You are back". Each circle depicts an operation (the non-mathematical ones are explained). Above each circle and to the right is the resulting vector we operate on at that step. We start out with looking up the embeddings corresponding to the words, concatenate them, and then proceed to multiply them with the respective layer weights, add the biases, and apply a reLU and a softmax function. After that, we compute the error by comparing to the true answer. Once we have computed the error, we can propagate it back through the graph.

computational graph

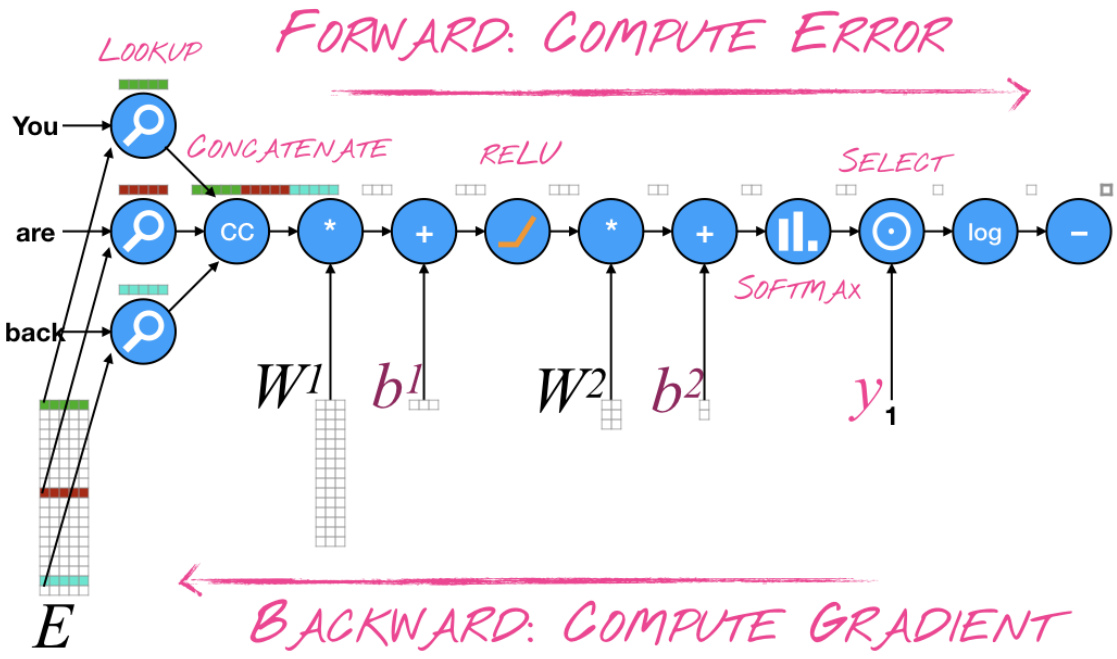


FIGURE 21. Computational graph.

Here, we can only provide a general introduction to the mechanics. For a much more detailed explanation, see Goldberg (2017).

13.6. Error Computation. In the perceptron, we updated the weights simply if the prediction did not match the true label. However, that approach works best for binary classification. In modern neural networks, we use more general ways to compute the error.

In order to do differential backpropagation, we need to first quantify *how far* we are off in our prediction. One of the most common ways to do so is **cross-entropy**.

cross-entropy

We compare the predicted probability distribution over the output labels (from a softmax) with the true answer. The true answer is encoded as a **one-hot** distribution, which is a probability distribution where all the mass is on one outcome, and all others are set to 0.0.

one-hot

$$-\sum_{k=1}^{K} \log(\hat{y}_k) * y_k$$

To compute the error, we take each output label k, multiply the logarithm of the predicted likelihood $\hat{y}_k$with the likelihood of the true answer y_k (i.e., either 0 or 1), sum all those numbers up, and take its negative.

13.7. Dropout.

> *"Dropout, simply described, is the concept that if you can learn how to do a task repeatedly whilst drunk, you should be able to do the task even better when sober." – Stephen Merity*

We have previously seen how regularization can improve the performance and generalizability of our classifiers. The idea is to make the training more difficult to fit, thereby forcing the model to find a more general explanation. Previously, we have done that by adding an error term that captures the sparsity and norm of the weights. We can still apply this form of regularization in neural networks, but there is another way that has proven effective.

In **dropout** Hinton et al. (2012), we simply remove a percentage of the nodes at random at each iteration. This forces the model to base its predictions on multiple paths in the network, rather than on one dominant node. **dropout**

The idea for regularization methods like dropout is motivated by overfitting examples like that of ALVINN Pomerleau (1989). ALVINN was a shallow network developed to steer an autonomous vehicle. It used input from cameras to learn to predict the right (re)action from the driver. After driving the car down a road, the researchers turned it around to drive back — and the car promptly went off the road. It turned out that there was a ditch on one side of the road, which showed up as darker pixels. ALVINN had learned to associate having those darker pixels on one side with keeping direction, so when it was turned around, the ditch was on the other side Pomerleau (2012). ALVINN had overfit to the ditch.

Methods like dropout essentially ask networks to do the job even if we take the helpful ditch away. It forces the network to pay attention to other factors (say, the side of the road).

In `keras`, we can add dropout as activation to any layer, specifying the proportion of nodes to affect.

```
1 dropout = Dropout(0.3, name='dropout')(layer)
```

CODE 72. Adding dropout to a layer in `keras`.

We will later see an example of dropout in use (see Code 77).

13.8. Data Preprocessing. In the following, we will use the `keras` library to implement networks. It provides two ways of specifying models: in a sequential manner or in a functional manner. In the sequential API, the sequence in which we specify the layers in the code corresponds to the structure of the network. In the functional API, each layer is a function that operates on the input we give it. The latter is a lot more

flexible, as we can add back input from a few layers further down. The examples in this book will all follow the functional API.

Keras requires some preprocessing of the data, to get it into the right format the library expects.

```
# collect known word tokens and tags
wordset, labelset = set(), set()

# collect all unique labels
labelset.update(set(train_labels))

# collect all word types in the training data
for words in train_instances:
    wordset.update(set(words))

# map words and tags into integer IDs
PAD = '-PAD-'
UNK = '-UNK-'
word2int = {word: i + 2 for i, word in enumerate(sorted(wordset))}
word2int[PAD] = 0  # special token for padding
word2int[UNK] = 1  # special token for unknown words

label2int = {label: i for i, label in enumerate(sorted(labelset))}
# structure to translate IDs back to labels
int2label = {i:label for label, i in label2int.items()}

# helper function
def convert2ints(instances):
    result = []
    for words in instances:
        # replace words with ID, use 1 for unknown words
        word_ints = [word2int.get(word, 1) for word in words]
        result.append(word_ints)
    return result

# convert data and labels
train_instances_int = convert2ints(train_instances)
train_labels_int = [label2int[label] for label in train_labels]

test_instances_int = convert2ints(test_instances)
test_labels_int = [label2int[label] for label in test_labels]

# convert labels to 1-hot encoding
from keras.utils import to_categorical
train_labels_1hot = to_categorical(train_labels_int, len(label2int))
test_labels_1hot = to_categorical(test_labels_int, len(label2int))
```

CODE 73. Data preprocessing for use in `keras`.

In theory, we can process input of any length with recurrent or convolutional neural networks. In practice, we need to specify a maximum length we expect to see. If we have instances that are longer, they simply get chopped off after that length. Shorter instances will get padded with a special 0 token, which is why we reserved a special token.

```
# compute 95th percentile of training sentence lengths
L = sorted(map(len, train_instances))
MAX_LENGTH = L[int(len(L) * 0.95)]

# apply padding
from keras.preprocessing.sequence import pad_sequences
train_instances_int = pad_sequences(train_instances_int, padding='
    post', maxlen=MAX_LENGTH)
test_instances_int = pad_sequences(test_instances_int, padding='post'
    , maxlen=MAX_LENGTH)
```

CODE 74. Computing a maximum length and padding instances up that amount.

14. Neural Architectures and Models

14.1. Convolutional Neural Nets. Multilayer perceptrons or feedforward neural networks are powerful, but they do have their limitations. The main one is that they can only process input of a fixed length: if we set up the architecture to expect an input vector of dimension 300, then we have to stick to this. This restriction is unproblematic for classifying words, since we can just use embeddings, but it gets tricky with documents. Of course, we could use document embeddings, but this brings us back to the special properties of language: compositionality and long-range dependencies. In a document embedding or discrete representation, "Not bad, actually quite good," and "Not good, actually quite bad" look very similar. They do mean very different things, though! We can also not capture long-range dependencies, for example in relation extraction. In a single vector representation, this is impossible. Instead, we need a way to take an arbitrarily long sequence of word vectors (usually embeddings), and gobble them up to produce a single output.

Not for the first time, the solution to this problem comes from computer vision. To transform an image into a manageable vector, we can move a small window over the image pixel by pixel, from top left to bottom right, and take a snap shot of each position. The result is a smaller set of snap shots. We can then repeat this process until we have the dimensionality we need.

channels

The image is a matrix, where each cell represents a pixel. This matrix is repeated several times, to represent the various **channels** (RGB and alpha), to get a 3-dimensional tensor. In NLP, we only have a matrix: each row is a word embedding (we could have different channels representing different versions of the sentence, for example translations, but here, we assume a single channel).

regions

Each of the windows, or **regions**, is a word n-gram, i.e., a smaller matrix. We can have several regions of different sizes (say, bigrams and trigrams), or several copies of the same region size (say, two trigram regions). Each of these smaller matrices is a

filter

filter. Let's say we have a trigram filter and slide it over the sentence: do we go word by word to get overlapping snapshots, or do we move the filter by three words at each

stride

step to get adjacent snapshots? This decision is captured in the **stride** length.

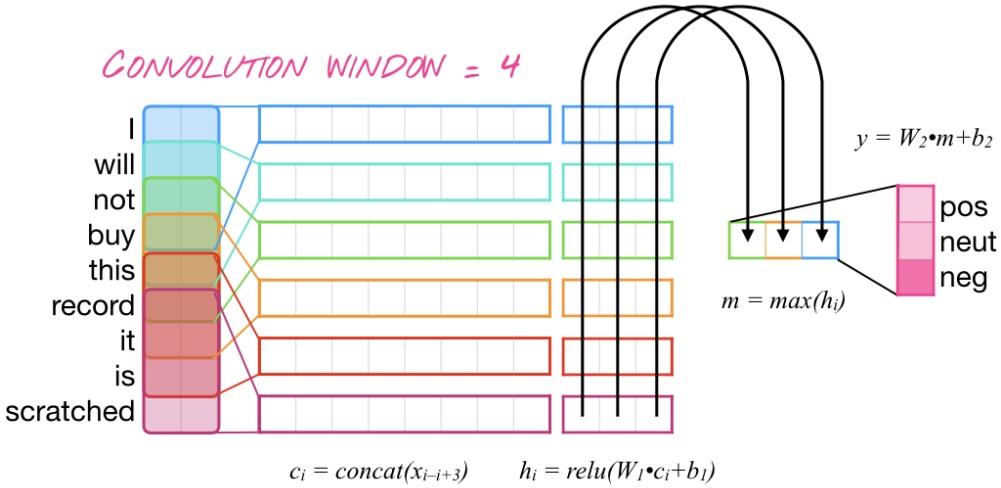


FIGURE 22. Text convolution with a filter size of four words and a stride of 1.

feature maps

Depending on the region size and the stride length, dragging the filter over the sentence produces a number of **feature maps**. For example, in Figure 22, we have one filter with a region size of four and stride length one. With nine input words, we get six feature maps. With four input words, we would have only one feature map. If we want, we can run each of these feature maps through an activation function.

We hope that each feature map captures some of the dependencies and compositions in the region, but it still leaves us with a variable output length. To reduce this variable

length into a fixed length, we use **pooling**. Simply put, we take the matrix of feature maps, and select from each of its dimensions the maximum value. The result is a fixed length vector: we have essentially squashed the feature map matrix into one row. Note that we could use other operations than the maximum (e.g., the minimum or mean). In practice, though, the maximum works best.

In `keras`, we can implement the various layers in a very straightforward manner:

```
from keras.models import Model
from keras.layers import Input
from keras.layers import Embedding
from keras.layers.convolutional import Conv1D
from keras.layers import GlobalMaxPooling1D, Dropout
from keras.layers.core import Dense, Activation
import numpy as np
np.random.seed(42)

# set parameters of matrices and convolution
embedding_dim = 64
nb_filter = 64
filter_length = 3

inputs = Input((MAX_LENGTH, ),
               name='word_IDs')
embeddings = Embedding(len(word2int),
                       embedding_dim,
                       input_length=MAX_LENGTH)(inputs)
convolution = Conv1D(filters=nb_filter,  # Number of filters
                     kernel_size=filter_length, #  stride length of
   each filter
                     padding='same',  #valid: don't go off edge; same:
    use padding before applying filter
                     activation='relu',
                     strides=1)(embeddings)
pooling = GlobalMaxPooling1D()(convolution)
dropout1 = Dropout(0.2)(pooling)
dense = Dense(32, activation='relu')(dropout1)
dropout2 = Dropout(0.2)(dense)
output = Dense(len(label2int), activation='softmax')(dropout2)

model = Model(inputs=[inputs], outputs=[output])
model.compile(optimizer='adam',
              loss='binary_crossentropy',
              metrics=['accuracy'])
model.summary()
```

CODE 75. `keras` implementation if a CNN.

We can now compile and run the model on the preprocessed data:

```
# batch size can have a huge effect on performance!
batch_size = 64
epochs = 5

history = model.fit(train_instances_int, train_labels_1hot,
                    batch_size=batch_size,
                    epochs=epochs,
                    verbose=1,
                    validation_data=(dev_instances_int,
   dev_labels_1hot)
                   )

loss, accuracy = model.evaluate(test_instances_int, test_labels_1hot,
                                batch_size=batch_size,
                                verbose=False)

print("\nTesting Accuracy:  {:.4f}".format(accuracy))
```

CODE 76. Compiling and fitting a CNN model in `keras`.

14.2. Recurrent Neural Nets. Recurrent neural networks are designed to deal with structured prediction (see above). They assume we have a sequence of inputs, and produce a sequence of outputs (usually, but not necessarily, the same number of inputs and outputs).

At their most generic form (see Figure 23), they have an input layer, a hidden layer, and an output layer. The hidden layer is a combination of the hidden state of the previous time step and the input at the current time step. The output is simply the result of applying an activation function to the hidden state. Concretely, we could make a linear combination of the input and hidden state, run it through a $tanh$ activation function to squash the result into the range between -1 and 1, and output that number. This approach would allow us to make a binary classification at each time step, depending on the previous words. For example, if we wanted to decide at each time step whether a word was English or a foreign word (i.e., code-switching).

One of the most useful applications of RNNs is in language modeling. Instead of predicting some label, we predict at each step which word of the vocabulary should come next (see Figure 24 for an example). This predicted word is then the input to the next time step, and so forth. In essence, an RNN language model feeds itself the next steps. And because we store the information of what has happened before in the hidden state, we can essentially keep an unlimited history around (encoded somehow in the

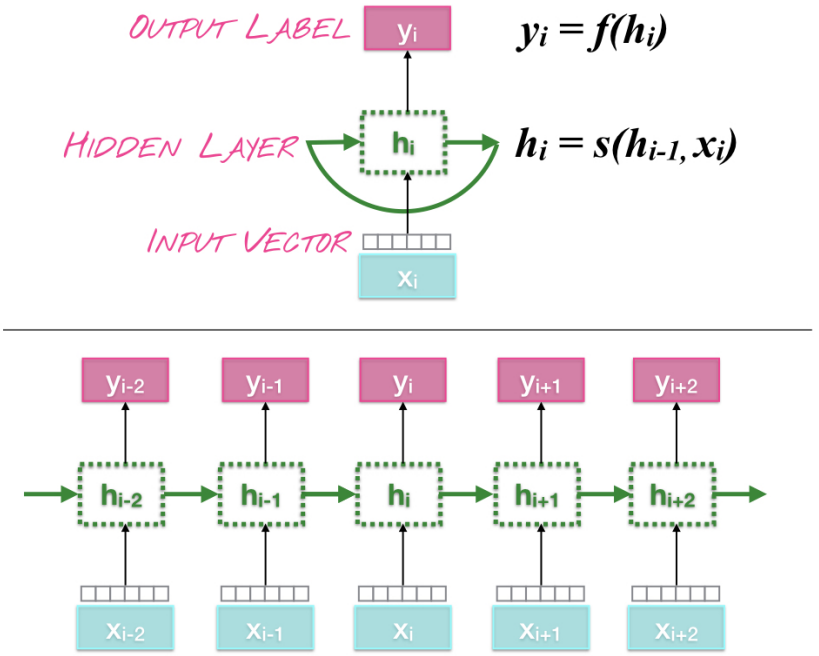


FIGURE 23. Generic recurrent neural network, in compact form (top) and unrolled (bottom), based on Goldberg (2017).

vector representation of the hidden state). In contrast, probabilistic language models need to limit themselves to a fixed context (the Markov order of the model). So they need to trade off how far back they can look with how many parameters they need. We could have a 15-gram probabilistic language model, but that would require us to keep a huge amount of very sparse n-gram probabilities around, which is not very practical.

When we speak, we not only base our next word on what have just said, but also on what we plan to say afterward (Levelt, 1993). It therefore makes sense to model this future-dependence in recurrent models. To do that, we just have to run a recurrent neural net on the reversed sentence: we start with the last word, and work our way backward. Each hidden state therefore encodes all the words that come after it. By combining the hidden states of this model with a regular RNN (usually by simply concatenating the two vectors), we get a **bi-directional RNN** (see Figure 25). **bi-directional RNN**

Having, in essence, an unlimited memory of the previous words (or the future ones, as we have just seen) is great in theory. In practice, though, it turns out to be not always useful. In all the examples of long-range dependencies we have seen so far, there was one word or phrase we needed to remember, but the intervening material was often unrelated and self-contained. So once we have processed an inserted subordinate clause,

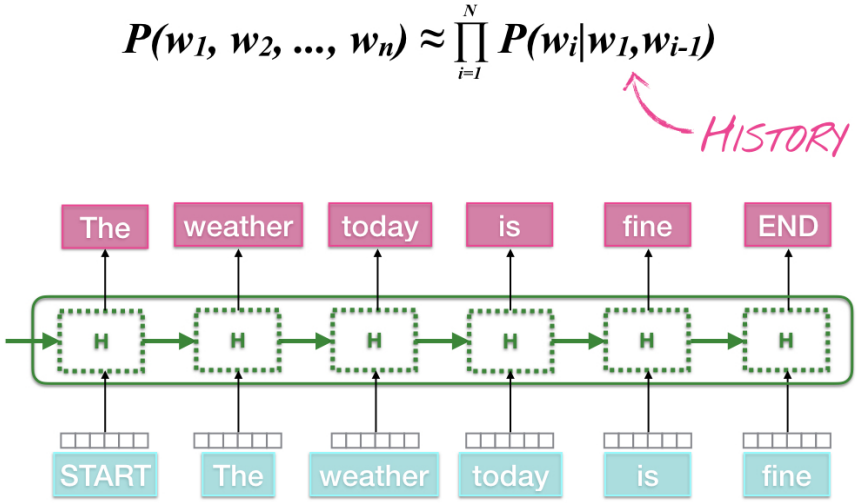


FIGURE 24. Example of a recurrent neural network language model, with theoretically infinite history.

we need no longer burden our memory with it anymore. All it does is take up space and potentially create confusion. The trick is to know what to keep and what to forget. This was the intuition behind the somewhat oddly named **long short-term memory (LSTM)** (Hochreiter and Schmidhuber, 1997). To achieve this selective memory, it includes various activation functions and gates (see Figure 26 for a schematic).

long short-term memory (LSTM)

In `keras`, we can relatively easily implement an LSTM or Bi-LSTM.

```
from keras.models import Model
from keras.layers import Input, Embedding
from keras.layers import Bidirectional, LSTM
from keras.layers import Dropout, Dense, Activation
import numpy as np

# Set a random seed for reproducibility
np.random.seed(42)

inputs = Input((MAX_LENGTH, ),
               name='word_IDs')
embeddings = Embedding(input_dim=len(word2int),
                       output_dim=128,
                       mask_zero=True,
```

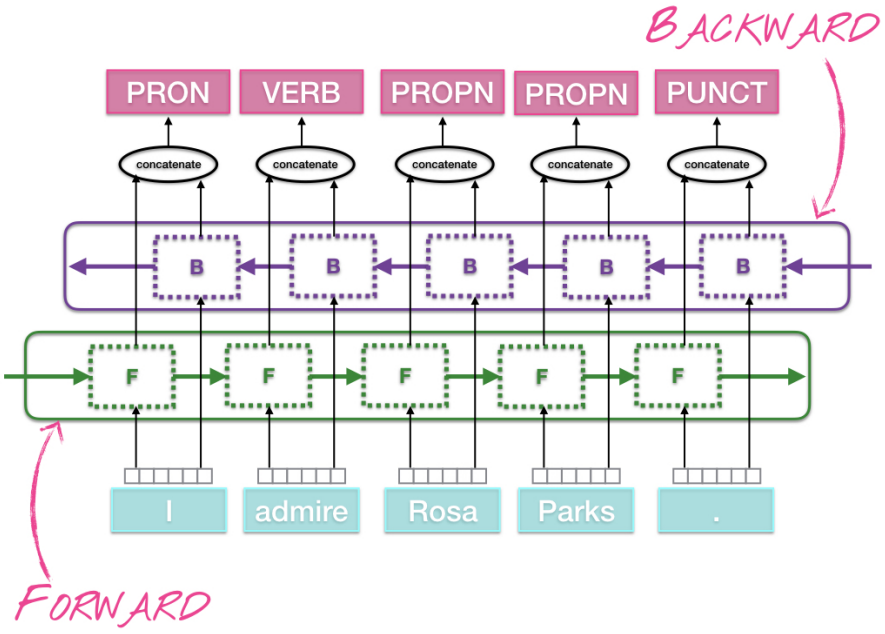


FIGURE 25. Bi-directional recurrent neural network architecture for POS tagging.

```
                          name='embeddings')(inputs)
lstm = LSTM(units=256,
               return_sequences=True,
               name="LSTM")(embeddings)
# for a Bi-LSTM, replace the line above with this:
# from keras.layers import Bidirectional
#bilstm = Bidirectional(LSTM(units=256,
#                            return_sequences=True),
#                        name="Bi-LSTM")(embeddings)
dropout = Dropout(0.3, name='dropout')(lstm)
lstm_out = Dense(len(tag2int), name='output')(dropout)
output = Activation('softmax', name='softmax')(lstm_out)

model = Model(inputs=[inputs], outputs=[output])
model.summary()
```

CODE 77. A (Bi-)LSTM in `keras`.

To train the model and log the output, we can use this code:

```
batch_size = 32
epochs = 5
```

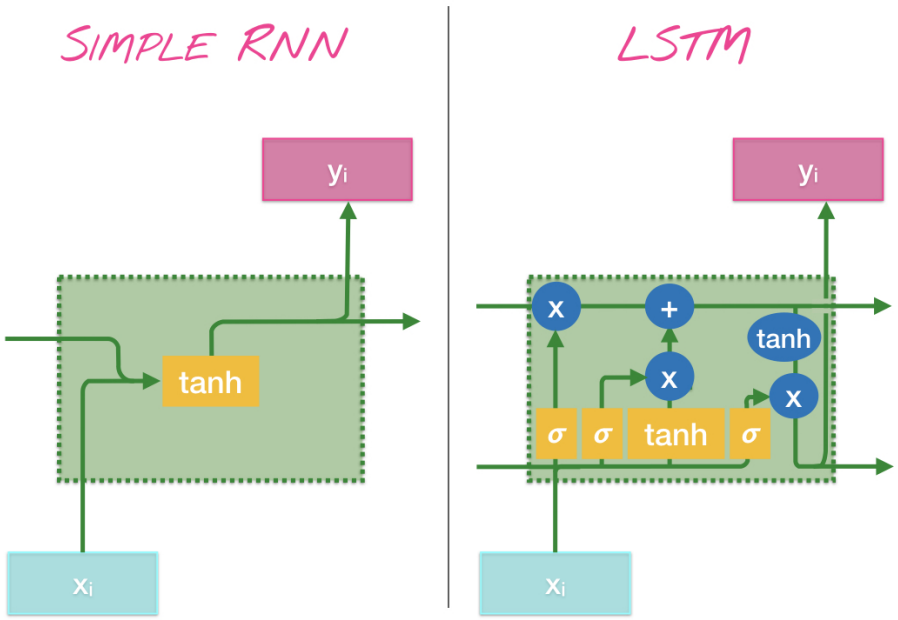


FIGURE 26. Internal structure of the simple RNN and the LSTM hidden state.

```
3
4  # compile the model we have defined above
5  model.compile(loss='categorical_crossentropy',
6                optimizer='adam',
7                metrics=['accuracy']
8               )
9
10 # run training and capture ouput log
11 history = model.fit(train_sentences, train_tags_1hot,
12                     batch_size=batch_size,
13                     epochs=epochs,
14                     verbose=1,
15                     validation_split=0.2)
```

CODE 78. Compiling and training a `keras` model.

attention *Thank you for your attention!*

14.3. Attention. We have repeatedly touched upon the importance of context, and the value of selective memory. It turns out that there is a way to combine these two aspects. The mechanism is called **attention**. It was first introduced in machine translation, where it is necessary to align the words in the sentences of the input and output languages. These alignments are not always one to one. For example, the English "not" corresponds to a "ne [...] pas" in French. The English dummy "do" (in questions) or

the impersonal German "es" (in generic statements) often have no corresponding word in other languages.

To capture these correspondences, Bahdanau et al. (2015) introduced the attention mechanism, later refined by Luong et al. (2015). It is essentially a heat map of the token correlations in both sentences. Higher values mean more correlated words. To compute a value a_{ij} in the attention matrix a, we first score the influence of each hidden states on the word j. We then normalize each hidden state vector i by its normalized score, and sum it up. By making this matrix a parameter of the model, the scoring is learned along with the other parameters of the network.

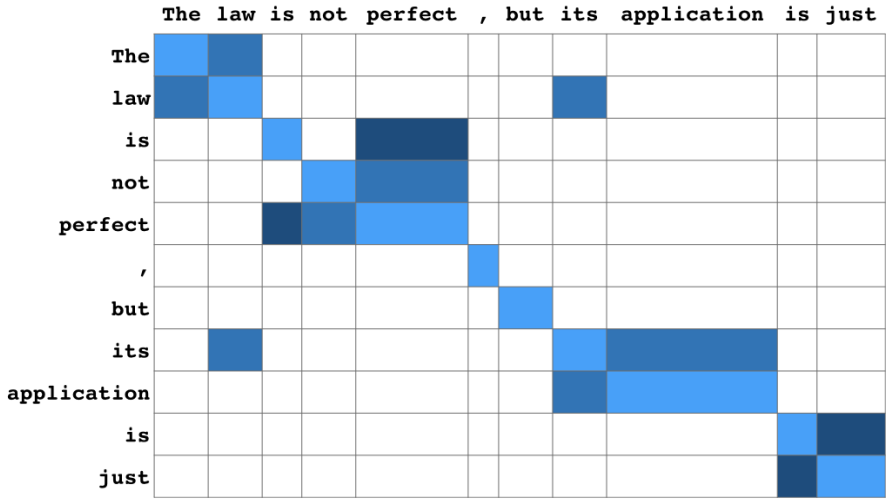


FIGURE 27. Self-attention in a CNN.

Attention works very well for word alignment of input and output in machine translation (see Figure 28), but it soon turned out to be useful for classification, by capturing complex expressions. When attention is applied to the input and a copy of itself, it captures long-term syntactic and semantic relations between words (see example in Figure 27). Adding this **self-attention** to CNNs made them better at classification, without increasing the parameter space too much. **self-attention**

14.4. The Transformer. For a while, it was unclear which architecture was better for text analysis, CNNs or RNNs. Much of the discussion focused on contextuality and long-range dependencies. Attention resolved this question, albeit by replacing both architectures with a third one: the **Transformer**. **Transformer**

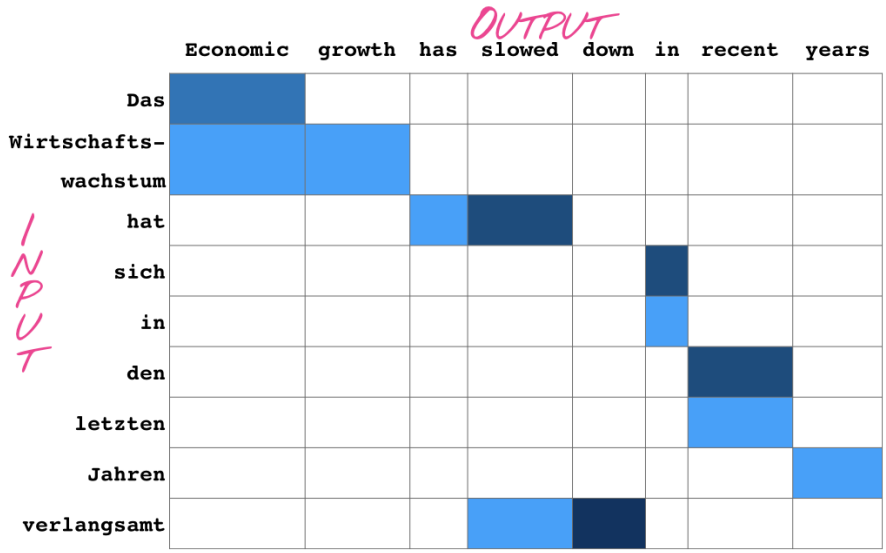


FIGURE 28. Attention in a RNN.

Vaswani et al. (2017) introduced the Transformer model. It combined the best of both worlds: selective long range-dependency of LSTMs, and the aggregate context of CNNs. As a result, Transformer-based architectures have dominated the leaderboards of various NLP tasks since. The architecture of the Transformer is like a series of Russian dolls: two parts with six elements each, where each of the first six has two parts (one of which is replicated eight times), and each of the second six has three elements.[8]

encoder
decoder

At the highest level, the Transformer consists of two components: an **encoder** and a **decoder** (see Figure 29). Both of them are made up of stacks of six smaller elements, which sequentially process the input sentence.[9]

Each of the encoder stack elements has two parts: a self-attention over the input, which is then fed into a simple feed-forward neural network. Self-attention is simply the correlation of all input words with all other input words. The self-attention is where most of the magic of the Transformer happens. We first create three hidden state "views"

query, key, and value

of the input, called the **query, key, and value** (each derived from the input via a dedicated weight set). We combine the key and query vectors, and compute the probability

[8]For an animated step-by-step visualization of the Transformer, see `http://jalammar.github.io/illustrated-transformer/`.

[9]The number six is arbitrary, and can be changed.

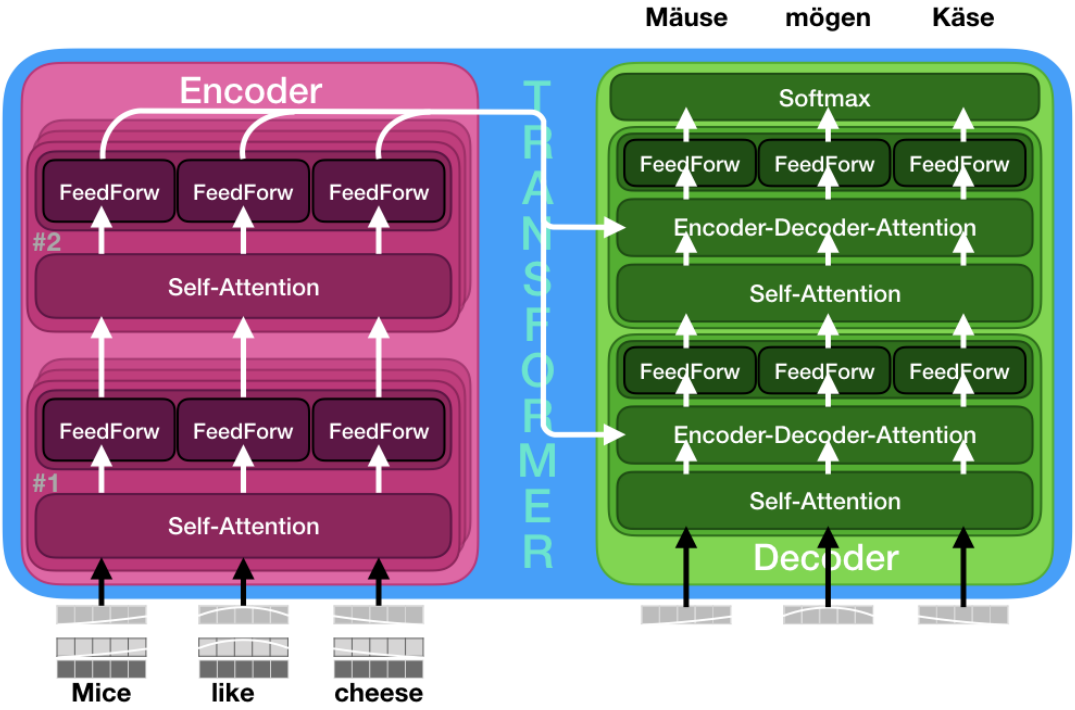


FIGURE 29. A schematic of the transformer.

distribution over the normalized sums, then multiply the result with the value vector (see Figure 30).

This process is repeated eight times in each layer,[10] once by each **attention head**. This has a similar effect as running several filters of the same region size in CNNs. Each attention head learns to attend to a different aspect of the data. For example, one might capture coreference relations between pronouns and their referents. Another one might capture verbal relations. The output of the eight heads is reduced to a single self-attention result before it is fed through a feedforward neural network, which produces an output for each word.

attention head

Self-attention and the multiple heads capture a lot of local context, but they are still local. I.e., they lose where in the sentence a word came from. In order to encode this positional component, the input embeddings are multiplied with a **positional encoding**. One of the ways in which this can be done is with a long sinusoidal wave.

positional encoding

The combination of self-attention with eight heads plus network is one of the six encoding elements. These elements are stacked, with each element receiving the output

[10] Again, this is number is an arbitrary choice.

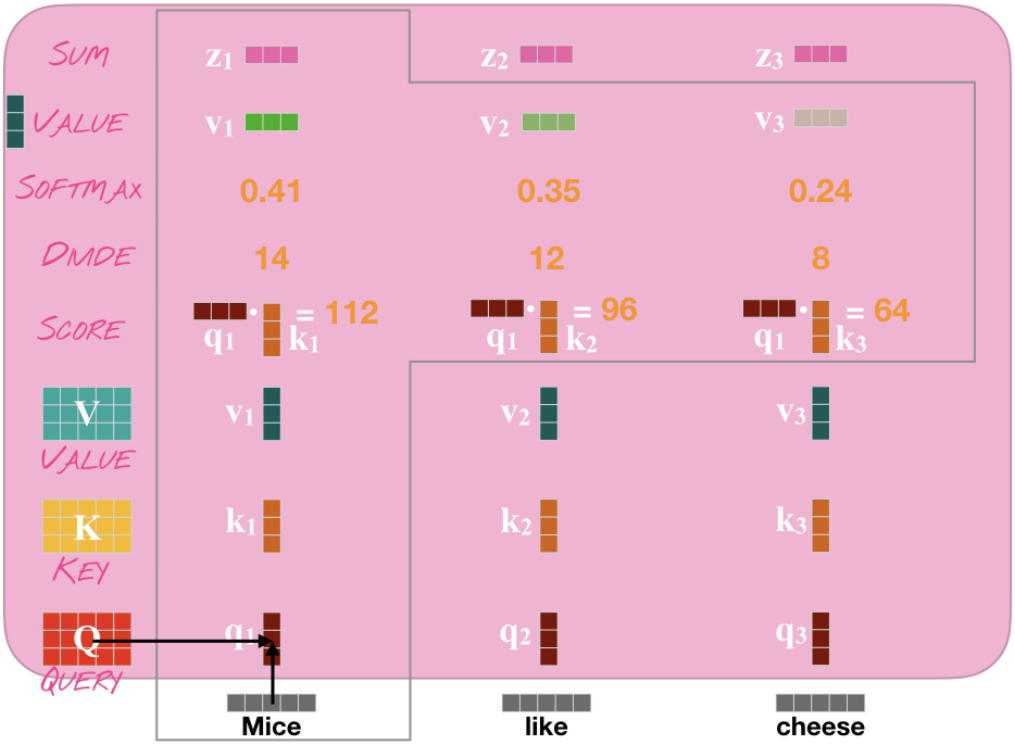


FIGURE 30. A simplified schematic of the Transformer encoder stack.

of the previous one. For the output of the last encoder, we again compute a key, and value vector, to produce an overall encoding representation.

This final encoding representation is passed on to each of the six decoders. (There are as many decoders as encoders.) In addition to the encoding input, each decoder receives the output of the previous decoding element. Each decoder again has several components: a self-attention layer, which is then combined with attention over the encoder element, and a feedforward network. The final decoder output is passed through a softmax layer for prediction.

The cost of the model is its complexity: in order to train the numerous parameters of a Transformer, we need sufficient data and computing power. While the former is becoming less of an issue, the latter has attracted some attention. The machines necessary for training have a substantial energy consumption, which impacts the environment (Strubell et al., 2019).

14.5. Neural Language Models. Neural networks also changed the field of language models, a class of **self-supervised** models. Each word serves as the input and as the output of the previous word. Traditionally, these models used probabilistic frameworks to compute how likely a sequence of words were. For computational feasibility, this involved computing the probability of each word in the limited context of the previous n words. I.e., we would break the probability of a sentence down into the product of the conditional probabilities of a word given its limited history of n previous words, $P(w_i|w_{i-n}, ..., w_{i-1})$.

self-supervised

Recurrent neural networks expanded this history-limitation to include potentially unlimited prior context. As a result, language models got a lot better. The Transformer also made a mark on this field. Using its best-of-both-worlds capabilities, a language model based on the Transformer captures both long-range dependencies and local coherence. The most famous Transformer-based language model is **GPT** (Radford et al., 2018). It was so convincing in producing realistic-looking text that the inventors did not release the full model initially, for fear of abuse. They eventually released a scaled-down version in GPT2 (Radford et al., 2019) and a companion piece discussing the implications (Solaiman et al., 2019).

GPT

Another Transformer-based language model is BERT (Devlin et al., 2019). BERT officially stands for *Bidirectional Encoder Representations from Transformers*, following an unexplained trend to name embedding models after Sesame Street characters. It uses the Transformer, but in both directions, to perform a cloze tasks (i.e., fill the blank). It takes as sentence in context and learns a representation encoding of it, while learning to predict randomly blanked out words (i.e., they are replaced with a `MASK` token). BERT uses a special classification token (`CLS`) at the beginning of each sentence, which represents the entire sentence, and is often used as input to classifiers. See Figure 31 for an example.

These representations have proven so adept at capturing semantic and syntactic information, that they have improved almost any task where they were used. This includes both NLP tasks and social science applications: Vicinanza et al. (2020) used a BERT-based measure of prescience to predict which ideas will become influential in a field. BERT's success has been so complete that a cottage-industry of papers dissecting BERT and its capabilities has emerged. Some people are now speaking (only half-joking) of BERTology (Rogers et al., 2020).

The original paper included a multilingual version, which allows training a model on one language where we have a lot of data (say, English), and applying it to another language (**zero-shot learning**). A wealth of language-specific versions has emerged since, which usually perform better for that particular language (Nozza et al., 2020), but of course do not facilitate cross-lingual learning.

zero-shot learning

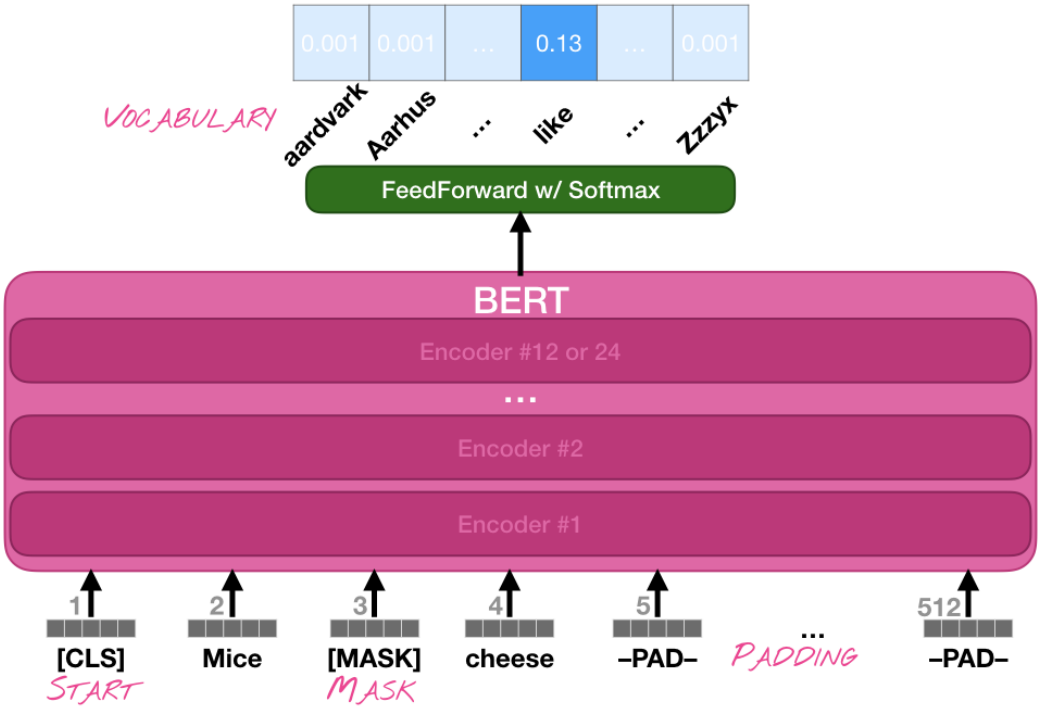


FIGURE 31. A schematic of BERT

All of these models have emerged in a very short amount of time, but have steadily increased in capabilities and performance. It is not clear what we will see in the next few years, but it is clear that we have not reached the end yet. There are still tasks to be solved, and we are only now starting to see applications to new areas of language understanding. These developments might upend everything written here, or they might build on it. Either way, I hope this book has enabled you to tackle them in stride!

Index

Bibliography

Adewole S Adamson and Avery Smith. 2018. Machine learning and health care disparities in dermatology. *JAMA dermatology*, 154(11):1247–1248.

Jalal S Alowibdi, Ugo A Buy, and Philip Yu. 2013. Empirical Evaluation of Profile Characteristics for Gender Classification on Twitter. In *Machine Learning and Applications (ICMLA), 2013 12th International Conference on*, volume 1, pages 365–369. IEEE.

Julia Angwin, Jeff Larson, Surya Mattu, and Lauren Kirchner. 2016. Machine bias. *ProPublica, May*, 23.

Maria Antoniak and David Mimno. 2018. Evaluating the stability of embedding-based word similarities. *Transactions of the Association for Computational Linguistics*, 6:107–119.

Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. 2014. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*.

Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. 2015. Neural machine translation by jointly learning to align and translate. In *3rd International Conference on Learning Representations, ICLR 2015*.

David Bamman, Brendan O'Connor, and Noah Smith. 2012. Censorship and deletion practices in Chinese social media. *First Monday*, 17(3).

Emily M Bender and Batya Friedman. 2018. Data Statements for Natural Language Processing: Toward Mitigating System Bias and Enabling Better Science. *Transactions of the Association for Computational Linguistics*, 6:587–604.

Yoshua Bengio, Réjean Ducharme, Pascal Vincent, and Christian Jauvin. 2003. A neural probabilistic language model. *Journal of machine learning research*, 3(Feb):1137–1155.

Taylor Berg-Kirkpatrick, David Burkett, and Dan Klein. 2012. An empirical investigation of statistical significance in NLP. In *Proceedings of the 2012 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning*, pages 995–1005. Association for Computational Linguistics.

Sudeep Bhatia. 2017. Associative judgment and vector space semantics. *Psychological review*, 124(1):1.

Federico Bianchi, Silvia Terragni, and Dirk Hovy. 2020. Pre-training is a Hot Topic: Contextualized Document Embeddings Improve Topic Coherence. *arXiv preprint arXiv:2004.03974*.

David M Blei, Andrew Y Ng, and Michael I Jordan. 2003. Latent Dirichlet Allocation. *Journal of Machine Learning Research*, 3(Jan):993–1022.

Su Lin Blodgett, Lisa Green, and Brendan O'Connor. 2016. Demographic Dialectal Variation in Social Media: A Case Study of African-American English. In *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing*, pages 1119–1130.

Tolga Bolukbasi, Kai-Wei Chang, James Y Zou, Venkatesh Saligrama, and Adam T Kalai. 2016. Man is to computer programmer as woman is to homemaker? debiasing word embeddings. In *Advances in neural information processing systems*, pages 4349–4357.

Jordan Boyd-Graber, David Mimno, and David Newman. 2014. *Care and Feeding of Topic Models: Problems, Diagnostics, and Improvements*, CRC Handbooks of Modern Statistical Methods. CRC Press, Boca Raton, Florida.

Kakia Chatsiou and Slava Jankin Mikhaylov. 2020. Deep learning for political science. *arXiv preprint arXiv:2005.06540*.

Stanley F. Chen and Joshua Goodman. 1996. An Empirical Study of Smoothing Techniques for Language Modeling. In *34th Annual Meeting of the Association for Computational Linguistics*.

Francois Chollet. 2017. *Deep learning with python*. Manning Publications Co.

Morgane Ciot, Morgan Sonderegger, and Derek Ruths. 2013. Gender Inference of Twitter Users in Non-English Contexts. In *Proceedings of the 2013 Conference on Empirical Methods in Natural Language Processing, Seattle, Wash*, pages 18–21.

Maximin Coavoux, Shashi Narayan, and Shay B Cohen. 2018. Privacy-preserving Neural Representations of Text. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 1–10.

Michael Collins. 2002. Discriminative Training Methods for Hidden Markov Models: Theory and Experiments with Perceptron Algorithms. In *Proceedings of the 2002 Conference on Empirical Methods in Natural Language Processing (EMNLP 2002)*, pages 1–8. Association for Computational Linguistics.

David Crystal. 2003. *The Cambridge Encyclopedia of the English Language*, 3rd edition. Cambridge University Press.

Rajarshi Das, Manzil Zaheer, and Chris Dyer. 2015. Gaussian LDA for topic models with word embeddings. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pages 795–804.

Scott Deerwester, Susan T Dumais, George W Furnas, Thomas K Landauer, and Richard Harshman. 1990. Indexing by latent semantic analysis. *Journal of the American society for information science*, 41(6):391–407.

Gary S Dell. 1986. A spreading-activation theory of retrieval in sentence production. *Psychological review*, 93(3):283.

Arthur P Dempster, Nan M Laird, and Donald B Rubin. 1977. Maximum likelihood from incomplete data via the EM algorithm. *Journal of the Royal Statistical Society: Series B (Methodological)*, 39(1):1–22.

Matthew J Denny and Arthur Spirling. 2018. Text preprocessing for unsupervised learning: why it matters, when it misleads, and what to do about it. *Political Analysis*, 26(2):168–189.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 4171–4186.

Adji B Dieng, Francisco JR Ruiz, and David M Blei. 2019. Topic modeling in embedding spaces. *arXiv preprint arXiv:1907.04907*.

Jacob Eisenstein. 2019. *Introduction to natural language processing*. Mit Press.

Yanai Elazar and Yoav Goldberg. 2018. Adversarial Removal of Demographic Attributes from Text Data. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 11–21.

James A Evans and Pedro Aceves. 2016. Machine translation: mining text for social theory. *Annual Review of Sociology*, 42:21–50.

Manaal Faruqui, Jesse Dodge, Sujay Kumar Jauhar, Chris Dyer, Eduard Hovy, and Noah A Smith. 2015. Retrofitting Word Vectors to Semantic Lexicons. In *Proceedings of the 2015 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 1606–1615.

John R Firth. 1957. A synopsis of linguistic theory, 1930-1955. *Studies in linguistic analysis*.

Karën Fort, Gilles Adda, and K. Bretonnel Cohen. 2011. Last words: Amazon Mechanical Turk: Gold mine or coal mine? *Computational Linguistics*, 37(2):413–420.

Victoria Fromkin, Robert Rodman, and Nina Hyams. 2018. *An introduction to language*. Cengage Learning.

Juri Ganitkevitch, Benjamin Van Durme, and Chris Callison-Burch. 2013. PPDB: The paraphrase database. In *Proceedings of the 2013 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 758–764.

Nikhil Garg, Londa Schiebinger, Dan Jurafsky, and James Zou. 2018. Word embeddings quantify 100 years of gender and ethnic stereotypes. *Proceedings of the National Academy of Sciences*, 115(16):E3635–E3644.

Matthew Gentzkow, Bryan T Kelly, and Matt Taddy. 2017. Text as data. Technical report, National Bureau of Economic Research.

Yoav Goldberg. 2016. A primer on neural network models for natural language processing. *Journal of Artificial Intelligence Research*, 57:345–420.

Yoav Goldberg. 2017. Neural network methods for natural language processing. *Synthesis Lectures on Human Language Technologies*, 10(1):1–309.

Yoav Goldberg and Omer Levy. 2014. word2vec Explained: deriving Mikolov et al.'s negative-sampling word-embedding method. *arXiv preprint arXiv:1402.3722*.

Daniel G Goldstein and Gerd Gigerenzer. 2002. Models of ecological rationality: the recognition heuristic. *Psychological review*, 109(1):75.

Edouard Grave, Piotr Bojanowski, Prakhar Gupta, Armand Joulin, and Tomas Mikolov. 2018. Learning Word Vectors for 157 Languages. In *Proceedings of the International Conference on Language Resources and Evaluation (LREC 2018)*.

Kevin T Greene, Baekkwan Park, and Michael Colaresi. 2019. Machine learning human rights and wrongs: How the successes and failures of supervised learning algorithms can inform the debate about information effects. *Political Analysis*, 27(2):223–230.

Justin Grimmer and Brandon M Stewart. 2013. Text as data: The promise and pitfalls of automatic content analysis methods for political texts. *Political analysis*, 21(3):267–297.

William L Hamilton, Jure Leskovec, and Dan Jurafsky. 2016. Diachronic Word Embeddings Reveal Statistical Laws of Semantic Change. In *Proceedings of the 54th*

Meeting of the Association for Computational Linguistics, pages 1489–1501. Association for Computational Linguistics.

Jochen Hartmann, Juliana Huppertz, Christina Schamp, and Mark Heitmann. 2018. Comparing automated text classification methods. *International Journal of Research in Marketing*.

Drew Harwell. 2018. The accent gap. Why some accents don't work on Alexa or Google Home. *The Washington Post*.

Joseph Henrich, Steven J Heine, and Ara Norenzayan. 2010. The weirdest people in the world? *Behavioral and brain sciences*, 33(2-3):61–83.

Geoffrey E Hinton, Nitish Srivastava, Alex Krizhevsky, Ilya Sutskever, and Ruslan R Salakhutdinov. 2012. Improving neural networks by preventing co-adaptation of feature detectors. In *Advances in neural information processing systems*.

Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long short-term memory. *Neural computation*, 9(8):1735–1780.

Jake M Hofman, Amit Sharma, and Duncan J Watts. 2017. Prediction and explanation in social systems. *Science*, 355(6324):486–488.

Dirk Hovy. 2010. An Evening with... EM. Technical report, University of Southern California.

Dirk Hovy. 2016. The Enemy in Your Own Camp: How Well Can We Detect Statistically-Generated Fake Reviews–An Adversarial Study. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics*. Association for Computational Linguistics.

Dirk Hovy, Taylor Berg-Kirkpatrick, Ashish Vaswani, and Eduard Hovy. 2013. Learning whom to trust with MACE. In *Proceedings of the 2013 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 1120–1130.

Dirk Hovy and Tommaso Fornaciari. 2018. Improving Author Attribute Prediction by Retrofitting Linguistic Representations with Homophily. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 671–677.

Dirk Hovy and Christoph Purschke. 2018. Capturing Regional Variation with Distributed Place Representations and Geographic Retrofitting. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 4383–4394.

Dirk Hovy, Afhsin Rahimi, Tim Baldwin, and Julian Brooke. 2019. Visualizing Regional Language Variation Across Europe on Twitter. In Stanley D. Brunn and

Roland Kehrein, editors, *Handbook of the Changing World Language Map*. Dordrecht: Springer.

Dirk Hovy and Anders Søgaard. 2015. Tagging performance correlates with author age. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 2: Short Papers)*, volume 2, pages 483–488.

Dirk Hovy and Shannon L Spruit. 2016. The social impact of natural language processing. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, volume 2, pages 591–598.

Hongzhao Huang, Zhen Wen, Dian Yu, Heng Ji, Yizhou Sun, Jiawei Han, and He Li. 2013. Resolving entity morphs in censored data. In *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, volume 1, pages 1083–1093.

Ashlee Humphreys and Rebecca Jen-Hui Wang. 2017. Automated text analysis for consumer research. *Journal of Consumer Research*, 44(6):1274–1306.

Jagadeesh Jagarlamudi, Hal Daumé III, and Raghavendra Udupa. 2012. Incorporating lexical priors into topic models. In *Proceedings of the 13th Conference of the European Chapter of the Association for Computational Linguistics*, pages 204–213. Association for Computational Linguistics.

Fred Jelinek and Robert Mercer. 1980. Interpolated estimation of Markov source parameters from sparse data. In *Proceedings Workshop Pattern Recognition in Practice*, pages 381–397.

Hans Jonas. 1984. *The Imperative of Responsibility: Foundations of an Ethics for the Technological Age*. (Original in German: Prinzip Verantwortung.) Chicago: University of Chicago Press.

Anna Jørgensen, Dirk Hovy, and Anders Søgaard. 2015. Challenges of studying and processing dialects in social media. In *Proceedings of the Workshop on Noisy User-generated Text*, pages 9–18.

Dan Jurafsky. 2014. *The language of food: A linguist reads the menu*. WW Norton & Company.

Dan Jurafsky and James H Martin. 2014. *Speech and language processing*, 3rd edition. Pearson London.

Slava Katz. 1987. Estimation of probabilities from sparse data for the language model component of a speech recognizer. *IEEE transactions on acoustics, speech, and signal processing*, 35(3):400–401.

Svetlana Kiritchenko and Saif Mohammad. 2018. Examining Gender and Race Bias in Two Hundred Sentiment Analysis Systems. In *Proceedings of the Seventh Joint Conference on Lexical and Computational Semantics*, pages 43–53.

Jakub Konečnỳ, H Brendan McMahan, Felix X Yu, Peter Richtárik, Ananda Theertha Suresh, and Dave Bacon. 2016. Federated learning: Strategies for improving communication efficiency. *arXiv preprint arXiv:1610.05492*.

Austin C Kozlowski, Matt Taddy, and James A Evans. 2018. The geometry of culture: Analyzing meaning through word embeddings. *arXiv preprint arXiv:1803.09288*.

Vivek Kulkarni, Rami Al-Rfou, Bryan Perozzi, and Steven Skiena. 2015. Statistically significant detection of linguistic change. In *Proceedings of the 24th International Conference on World Wide Web*, pages 625–635. International World Wide Web Conferences Steering Committee.

William Labov. 1972. *Sociolinguistic patterns*. University of Pennsylvania Press.

Thomas K Landauer and Susan T Dumais. 1997. A solution to Plato's problem: The latent semantic analysis theory of acquisition, induction, and representation of knowledge. *Psychological review*, 104(2):211.

Serge Lang. 2012. *Introduction to linear algebra*. Springer Science & Business Media.

Jey Han Lau and Timothy Baldwin. 2016. An Empirical Evaluation of doc2vec with Practical Insights into Document Embedding Generation. page 78.

Quoc Le and Tomas Mikolov. 2014. Distributed representations of sentences and documents. In *Proceedings of the 31st International Conference on Machine Learning (ICML-14)*, pages 1188–1196.

Willem JM Levelt. 1993. *Speaking: From intention to articulation*, volume 1. MIT press.

Gideon Lewis-Kraus. 2016. The great AI awakening. *The New York Times Magazine*, 14.

Yitong Li, Timothy Baldwin, and Trevor Cohn. 2018. Towards Robust and Privacy-preserving Text Representations. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, volume 2, pages 25–30.

Wendy Liu and Derek Ruths. 2013. What's in a name? Using first names as features for gender inference in Twitter. In *Analyzing Microtext: 2013 AAAI Spring Symposium*.

Edward Loper and Steven Bird. 2002. NLTK: The Natural Language Toolkit. In *Proceedings of the ACL-02 Workshop on Effective Tools and Methodologies for Teaching Natural Language Processing and Computational Linguistics*.

Thang Luong, Hieu Pham, and Christopher D. Manning. 2015. Effective Approaches to Attention-based Neural Machine Translation. In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*, pages 1412–1421, Lisbon, Portugal. Association for Computational Linguistics.

Laurens van der Maaten and Geoffrey Hinton. 2008. Visualizing data using t-SNE. *Journal of machine learning research*, 9(Nov):2579–2605.

Christopher D Manning. 2015. Computational linguistics and deep learning. *Computational Linguistics*, 41(4):701–707.

Christopher D Manning and Hinrich Schütze. 1999. *Foundations of statistical natural language processing*. MIT press.

Stephen Marsland. 2011. *Machine learning: an algorithmic perspective*. Chapman and Hall/CRC.

Stephen Marsland. 2015. *Machine learning: an algorithmic perspective*, 2nd edition. Chapman and Hall/CRC.

Ryan McDonald, Joakim Nivre, Yvonne Quirmbach-Brundage, Yoav Goldberg, Dipanjan Das, Kuzman Ganchev, Keith Hall, Slav Petrov, Hao Zhang, Oscar Täckström, et al. 2013. Universal dependency annotation for multilingual parsing. In *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, volume 2, pages 92–97.

Nicolai Meinshausen and Peter Bühlmann. 2010. Stability selection. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, 72(4):417–473.

Tomáš Mikolov, Martin Karafiát, Lukáš Burget, Jan Černockỳ, and Sanjeev Khudanpur. 2010. Recurrent neural network based language model. In *Eleventh annual conference of the international speech communication association*.

Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. 2013. Distributed representations of words and phrases and their compositionality. In *Advances in neural information processing systems*, pages 3111–3119.

George A Miller. 1995. WordNet: a lexical database for English. *Communications of the ACM*, 38(11):39–41.

Sara Mills. 2012. *Gender matters: Feminist linguistic analysis*. Equinox Pub.

David Mimno, Hanna Wallach, Edmund Talley, Miriam Leenders, and Andrew McCallum. 2011. Optimizing Semantic Coherence in Topic Models. In *Proceedings of the 2011 Conference on Empirical Methods in Natural Language Processing*, pages 262–272.

Marvin Minsky and Seymour A Papert. 1969. *Perceptrons*. MIT press.

Ehsan Mohammady and Aron Culotta. 2014. Using county demographics to infer attributes of twitter users. In *Proceedings of the joint workshop on social dynamics and personal attributes in social media*, pages 7–16.

Mehryar Mohri, Afshin Rostamizadeh, and Ameet Talwalkar. 2018. *Foundations of machine learning*. MIT press.

Frederick Mosteller and David L Wallace. 1963. Inference in an authorship problem: A comparative study of discrimination methods applied to the authorship of the disputed Federalist Papers. *Journal of the American Statistical Association*, 58(302):275–309.

Robert Munro. 2013. NLP for all languages. Idibon Blog, May 22 `http://idibon.com/nlp-for-all` Retrieved May 17, 2016.

Kevin P Murphy. 2012. *Machine learning: a probabilistic perspective*. MIT Press.

Dong Nguyen, Noah A Smith, and Carolyn P Rosé. 2011. Author age prediction from text using linear regression. In *Proceedings of the 5th ACL-HLT Workshop on Language Technology for Cultural Heritage, Social Sciences, and Humanities*, pages 115–123. Association for Computational Linguistics.

Vlad Niculae, Srijan Kumar, Jordan Boyd-Graber, and Cristian Danescu-Niculescu-Mizil. 2015. Linguistic Harbingers of Betrayal: A Case Study on an Online Strategy Game. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, volume 1, pages 1650–1659.

Joakim Nivre, Željko Agić, Maria Jesus Aranzabe, Masayuki Asahara, Aitziber Atutxa, Miguel Ballesteros, John Bauer, Kepa Bengoetxea, Riyaz Ahmad Bhat, Cristina Bosco, et al. 2015. Universal Dependencies 1.2.

Joakim Nivre, Marie-Catherine de Marneffe, Filip Ginter, Yoav Goldberg, Jan Hajič, Christopher D. Manning, Ryan McDonald, Slav Petrov, Sampo Pyysalo, Natalia Silveira, Reut Tsarfaty, and Daniel Zeman. 2016. Universal Dependencies v1: A Multilingual Treebank Collection. In *Proceedings of the Tenth International Conference on Language Resources and Evaluation (LREC'16)*, pages 1659–1666, Portorož, Slovenia. European Language Resources Association (ELRA).

Debora Nozza, Federico Bianchi, and Dirk Hovy. 2020. What the [MASK]? Making Sense of Language-Specific BERT Models. *arXiv preprint arXiv:2003.02912*.

Cathy O'Neil. 2016. The Ethical Data Scientist. Slate, February 4 `http://www.slate.com/articles/technology/future_tense/2016/02/how_to_bring_better_ethics_to_data_science.html` Retrieved Feb 24, 2016.

Gregory Park, H Andrew Schwartz, Johannes C Eichstaedt, Margaret L Kern, Michal Kosinski, David J Stillwell, Lyle H Ungar, and Martin EP Seligman. 2015. Automatic personality assessment through social media language. *Journal of personality and social psychology*, 108(6):934.

Rebecca J. Passonneau and Bob Carpenter. 2014. The benefits of a model of annotation. *Transactions of the Association for Computational Linguistics*, 2:311–326.

Silviu Paun, Bob Carpenter, Jon Chamberlain, Dirk Hovy, Udo Kruschwitz, and Massimo Poesio. 2018. Comparing Bayesian Models of Annotation. *Transactions of the Association for Computational Linguistics*, 6:571–585.

Ellie Pavlick, Matt Post, Ann Irvine, Dmitry Kachaev, and Chris Callison-Burch. 2014. The language demographics of Amazon Mechanical Turk. *Transactions of the Association for Computational Linguistics*, 2:79–92.

James W. Pennebaker. 2011. *The Secret Life of Pronouns: What Our Words Say About Us*. Broadway: Bloomsbury Press.

James W Pennebaker, Martha E Francis, and Roger J Booth. 2001. Linguistic inquiry and word count: LIWC 2001. *Mahway: Lawrence Erlbaum Associates*, 71(2001):2001.

Jeffrey Pennington, Richard Socher, and Christopher D Manning. 2014. Glove: Global vectors for word representation. In *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*, pages 1532–1543.

Andrew Peterson and Arthur Spirling. 2018. Classification accuracy as a substantive quantity of interest: Measuring polarization in westminster systems. *Political Analysis*, 26(1):120–128.

Slav Petrov, Dipanjan Das, and Ryan McDonald. 2011. A universal part-of-speech tagset. In *Proceedings of LREC*.

Barbara Plank, Dirk Hovy, and Anders Søgaard. 2014. Learning part-of-speech taggers with inter-annotator agreement loss. In *Proceedings of the 14th Conference of the European Chapter of the Association for Computational Linguistics*, pages 742–751.

Dean A Pomerleau. 1989. Alvinn: An autonomous land vehicle in a neural network. In *Advances in neural information processing systems*, pages 305–313.

Dean A Pomerleau. 2012. *Neural network perception for mobile robot guidance*, volume 239. Springer Science & Business Media.

Martin F Porter. 1980. An algorithm for suffix stripping. *Program*, 14(3):130–137.

Vinodkumar Prabhakaran, Owen Rambow, and Mona Diab. 2012. Predicting overt display of power in written dialogs. In *Proceedings of the 2012 Conference of the North*

American Chapter of the Association for Computational Linguistics: Human Language Technologies, pages 518–522. Association for Computational Linguistics.

Daniel Preoţiuc-Pietro, Vasileios Lampos, and Nikolaos Aletras. 2015a. An analysis of the user occupational class through Twitter content. In *ACL*.

Daniel Preoţiuc-Pietro, Svitlana Volkova, Vasileios Lampos, Yoram Bachrach, and Nikolaos Aletras. 2015b. Studying user income through language, behaviour and affect in social media. *PloS one*, 10(9):e0138717.

Christoph Purschke and Dirk Hovy. 2019. Lörres, Möppes, and the Swiss. (Re)Discovering Regional Patterns in Anonymous Social Media Data. *Journal of Linguistic Geography*.

Alec Radford, Karthik Narasimhan, Tim Salimans, and Ilya Sutskever. 2018. Improving language understanding by generative pre-training. *URL https://s3-us-west-2. amazonaws. com/openai-assets/researchcovers/languageunsupervised/language understanding paper. pdf*.

Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. 2019. Language models are unsupervised multitask learners. *OpenAI Blog*, 1(8):9.

Philip Resnik and Eric Hardisty. 2010. Gibbs sampling for the uninitiated. Technical report, Maryland Univ College Park Inst for Advanced Computer Studies.

Margaret E Roberts, Brandon M Stewart, Dustin Tingley, Edoardo M Airoldi, et al. 2013. The structural topic model and applied social science. In *Advances in neural information processing systems workshop on topic models: computation, application, and evaluation*, pages 1–20. Harrahs and Harveys, Lake Tahoe.

Michael Röder, Andreas Both, and Alexander Hinneburg. 2015. Exploring the space of topic coherence measures. In *Proceedings of the eighth ACM international conference on Web search and data mining*, pages 399–408.

Phillip Rogaway. 2015. The Moral Character of Cryptographic Work. Technical report, IACR-Cryptology ePrint Archive.

Anna Rogers, Olga Kovaleva, and Anna Rumshisky. 2020. A Primer in BERTology: What we know about how BERT works. *arXiv preprint arXiv:2002.12327*.

Xin Rong. 2014. word2vec parameter learning explained. *arXiv preprint arXiv:1411.2738*.

Frank Rosenblatt. 1958. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, 65(6):386.

Sara Rosenthal and Kathleen McKeown. 2011. Age prediction in blogs: A study of style, content, and online behavior in pre-and post-social media generations. In *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies-Volume 1*, pages 763–772. Association for Computational Linguistics.

Rachel Rudinger, Jason Naradowsky, Brian Leonard, and Benjamin Van Durme. 2018. Gender Bias in Coreference Resolution. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 2 (Short Papers)*, volume 2, pages 8–14.

Maarten Sap, Dallas Card, Saadia Gabriel, Yejin Choi, and Noah A. Smith. 2019. The risk of racial bias in hate speech detection. In *Proceedings of the 57th Conference of the Association for Computational Linguistics*, pages 1668–1678, Florence, Italy. Association for Computational Linguistics.

Tyler Schnoebelen. 2013. The weirdest languages. Idibon Blog, June 21 `http://idibon.com/the-weirdest-languages` Retrieved May 17, 2016.

H Andrew Schwartz, Johannes Eichstaedt, Eduardo Blanco, Lukasz Dziurzynski, Margaret Kern, Stephanie Ramones, Martin Seligman, and Lyle Ungar. 2013. Choosing the right words: Characterizing and reducing error of the word count approach. In *Second Joint Conference on Lexical and Computational Semantics (* SEM), Volume 1: Proceedings of the Main Conference and the Shared Task: Semantic Textual Similarity*, pages 296–305.

Galit Shmueli et al. 2010. To explain or to predict? *Statistical science*, 25(3):289–310.

Rion Snow, Brendan O'Connor, Daniel Jurafsky, and Andrew Ng. 2008. Cheap and fast – but is it good? evaluating non-expert annotations for natural language tasks. In *Proceedings of the 2008 Conference on Empirical Methods in Natural Language Processing*, pages 254–263, Honolulu, Hawaii. Association for Computational Linguistics.

Irene Solaiman, Miles Brundage, Jack Clark, Amanda Askell, Ariel Herbert-Voss, Jeff Wu, Alec Radford, and Jasmine Wang. 2019. Release strategies and the social impacts of language models. *arXiv preprint arXiv:1908.09203*.

Karen Spärck Jones. 1972. A statistical interpretation of term specificity and its application in retrieval. *Journal of documentation*, 28(1):11–21.

Akash Srivastava and Charles Sutton. 2017. Autoencoding variational inference for topic models. *arXiv preprint arXiv:1703.01488*.

Keith Stevens, Philip Kegelmeyer, David Andrzejewski, and David Buttler. 2012. Exploring Topic Coherence over Many Models and Many Topics. In *Proceedings of the 2012 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning*, pages 952–961, Jeju Island, Korea. Association for Computational Linguistics.

Emma Strubell, Ananya Ganesh, and Andrew McCallum. 2019. Energy and policy considerations for deep learning in NLP. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pages 3645–3650, Florence, Italy. Association for Computational Linguistics.

Cass R Sunstein. 2004. Precautions Against What? The Availability Heuristic and Cross-Cultural Risk Perceptions. *U Chicago Law & Economics, Olin Working Paper*, (220):04–22.

Rachael Tatman. 2017. Gender and Dialect Bias in YouTube's Automatic Captions. In *Proceedings of the First ACL Workshop on Ethics in Natural Language Processing*, pages 53–59.

Joel Tetreault, Jill Burstein, and Claudia Leacock. 2015. *Proceedings of the Tenth Workshop on Innovative Use of NLP for Building Educational Applications*, chapter Proceedings of the Tenth Workshop on Innovative Use of NLP for Building Educational Applications. Association for Computational Linguistics.

Peter Trudgill. 2000. *Sociolinguistics: An introduction to language and society*. Penguin UK.

Amos Tversky and Daniel Kahneman. 1973. Availability: A heuristic for judging frequency and probability. *Cognitive psychology*, 5(2):207–232.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Advances in neural information processing systems*, pages 5998–6008.

Paul Vicinanza, Amir Goldberg, and Sameer Srivastava. 2020. Who sees the future? a deep learning language model demonstrates the vision advantage of being small. *SocArXiv. May*, 26.

Svitlana Volkova, Yoram Bachrach, Michael Armstrong, and Vijay Sharma. 2015. Inferring Latent User Properties from Texts Published in Social Media (Demo). In *Proceedings of the Twenty-Ninth Conference on Artificial Intelligence (AAAI)*, Austin, TX.

Svitlana Volkova, Glen Coppersmith, and Benjamin Van Durme. 2014. Inferring User Political Preferences from Streaming Communications. In *Proceedings of the 52nd annual meeting of the ACL*, pages 186–196.

Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, et al. 2016. Google's neural machine translation system: Bridging the gap between human and machine translation. *arXiv preprint arXiv:1609.08144*.

Tal Yarkoni and Jacob Westfall. 2017. Choosing prediction over explanation in psychology: Lessons from machine learning. *Perspectives on Psychological Science*, 12(6):1100–1122.

Mark Yatskar, Luke Zettlemoyer, and Ali Farhadi. 2016. Situation recognition: Visual semantic role labeling for image understanding. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 5534–5542.

Jieyu Zhao, Tianlu Wang, Mark Yatskar, Vicente Ordonez, and Kai-Wei Chang. 2017. Men Also Like Shopping: Reducing Gender Bias Amplification using Corpus-level Constraints. In *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing*, pages 2979–2989.

George K Zipf. 1935. The psycho-biology of language: an introduction to dynamic philology.

1. Probabilities

In some cases, we will represent words as probabilities. In the case of topic models or language models, this allows us to reason under uncertainty. However, there is an even simpler reason to use probabilities: Keeping track of frequencies is often not too practical for our purposes. In one of our previous examples, some quarterly reports might be much longer than others, so seeing "energy" mentioned more often in those longer reports than in shorter ones is not too informative by itself. We need to somehow account for the length of the documents.

For this, normalized counts, or probabilities, are a much better indicator. Of course, probabilities are nothing more than counts normalized by a **Z-score**. Typically, those Z-scores are the counts of all words in our corpus. **Z-score**

$$P(w) = \frac{count(w)}{N}$$

where N is the total count of words in our corpus.

A **probability distribution** over a vocabulary therefore assigns a probability between 0.0 and 1.0 to each word in the vocabulary. And if we summed up all probabilities in the vocabulary, the result would be 1.0 (otherwise, it is not a probability distribution). **probability distribution**

Technically, probabilities are continuous values. However, since each probability is a discrete feature (i.e., it "means" something) in a feature vector, we cover them here under discrete representations.

Say we have a corpus of 1000 documents, x is "natural" and occurs in 20 documents, and y is "language" and occurs in 50 documents.

The probability $\mathbf{P}(\mathbf{x})$ of seeing a word x if we reached into a big bag with all words in our vocabulary, is therefore simply the number of documents that contain x, say "natural" (here, 20), divided by the number of all documents in our corpus (1000). $\mathbf{P}(\mathbf{x})$

$$P(natural) = \frac{count(documents\ w.\ \text{“}natural\text{”})}{number\ of\ all\ documents} = \frac{20}{1000} = \frac{1}{50} = 0.02$$

$$P(language) = \frac{count(documents\ w.\ \text{“}language\text{”})}{number\ of\ all\ documents} = \frac{50}{1000} = \frac{1}{20} = 0.05$$

Note that probabilities are not percentages (even though they are often used that way in everyday use)! In order to express a probability in percent, you need to multiply it by 100. We therefore have a 2% probability of randomly drawing the word "natural", and a 5% chance of drawing "language".

As we have seen, though, word frequencies follow a Zipf distribution, so the most frequent words get a lot of the probability mass, whereas most of the words get a very small probability. Since this can lead to vanishingly small numbers that computers sometimes can not handle, it is common to take the logarithm of the probabilities (also called **log-probabilities**). Note that in log-space, multiplication becomes addition, **log-probabilities**

and division becomes subtraction. Therefore, an alternative way to compute the (log) probability of a word is to subtract the log-count of the total number of words from the log-count of the word count.

$$\log P(w) = \log count(w) - \log N$$

To get a normal probability back, we have to exponentiate, but this can cause a loss of precision. It is therefore usually better to store raw counts and convert them to log-probabilities as needed.

$$\log P(natural) = \log 20 - \log 1000 = \log 1 - \log 50 = -3.912023005428146$$

joint probability

1.1. Joint Probability. $P(x, y)$ is a **joint probability**, i.e., how likely is it that we see x and y together, that is the words in one document ("natural language", "language natural", or separated by other words, since order does not matter in this case). Say we have 10 documents that contain both words, then

$$P(natural, language) = \frac{count(documents\ w.\ \text{“}natural\text{”}\ and\ \text{“}language\text{”})}{number\ of\ all\ documents}$$

$$= \frac{10}{1000} = \frac{1}{100} = 0.01$$

1.2. Conditional Probability. Words do not occur in isolation, but in context, and in certain contexts, they are more likely than in others. If you hear the sentence "Most of the constituents mistrusted the ..." you expect only a small number of words to follow. "Politician" is much more likely than, say "handkerchief". That is the probability of "politician" in this context is much higher than that of "handkerchief". Or, to put it in mathematical notation:

$$P(politician|CONTEXT) > P(handkerchief|CONTEXT)$$

conditional probability

$P(y|x)$ is a **conditional probability**, i.e., how likely are we to see y after having seen x.

Let's reduce the context to one word for now and look at the words in our previous example, i.e., "language" in a document that contains "natural". We can compute this as

$$P(language|natural) = \frac{P(x, y)}{P(x)} = \frac{\frac{count(documents\ w.\ \text{“}natural\text{”}\ and\ \text{“}language\text{”})}{number\ of\ all\ documents}}{\frac{count(documents\ w.\ \text{“}natural\text{”})}{number\ of\ all\ documents}}$$

$$= \frac{count(documents\ w.\ \text{“}natural\text{”}\ and\ \text{“}language\text{”})}{count(documents\ w.\ \text{“}natural\text{”})}$$

$$= \frac{10}{20} = \frac{1}{2} = 0.5$$

Note that order *does* matter in this case! So the corresponding probability formulation for seeing "natural" in a document that contains "language", or $P(x|y) = \frac{P(x,y)}{P(y)}$ is something completely different (namely 0.2).

1.3. Probability Distributions. So far, we have treated probabilities as normalized counts, and for the most part, that works well. However, there is a more general way of looking at probabilities, and that is as functions. They take as input an event (a word or some other thing), and returns a number between 0 and 1. In the case of normalized counts, the function itself was a lookup table, or maybe a division. However, we can also have probability functions that are defined by an equation, and that take a number of parameters other than the event we are interested in.

The general form of these probability functions is

$$f(x; \theta)$$

where x is the event we are interested in, and θ is just a shorthand for any of the additional parameters.

What makes these functions probability distributions are the facts that they

(1) cover the entire sample space (for example all words)
(2) sum to 1.0 (if we added up the probabilities of all words)

discrete probability distributions

In the case of texts, we mostly work with **discrete probability distributions**, i.e., there is a defined number of events, for example words. When we graph these distributions, we usually use bar graphs. There are also continuous probability distributions, the most well known of which is the normal or Gaussian distribution. However, for the sake of this book, we will not need them. In the following, we will look at some of the most common discrete probability distributions.

1.3.1. *Uniform distribution.* The simplest distribution is the one where all events are equally likely, so all we have to return is 1 divided by the number of outcomes.

$$U(x; N) = \frac{1}{N}$$

A common example is a die roll: all numbers are equally likely to come up. This is also a good distribution to use if we do not know anything about the data and do not want to bias our model unduly.

1.3.2. *Bernoulli distribution.* This is the simplest possible discrete distribution: it only has two outcomes, and can be described with a single parameter q, which is the chance of success. The other outcome is simply $1 - q$

$$Pr(x; q) = \begin{cases} 1 - q & if x = 0 \\ q & if x = 1 \end{cases}$$

If $q = 0.5$, the Bernoulli distribution is also a uniform distribution.

1.3.3. *Multinomial distribution.* Most often, we will deal with distributions that are much larger than two outcomes, and much more irregular than uniform. Formally, it is parametrized by a single parameter θ, which is a vector with all probabilities. The mathematical definition is

$$P(x;\theta) = \prod_{j=1}^{1} \theta_j^{I(x_j)=1}$$

which is a very complicated way of writing "use the value at the vector position for x"

A good example is the distribution over characters in a sample of text. In Figure 32, we can see how different characters have very different probabilities.

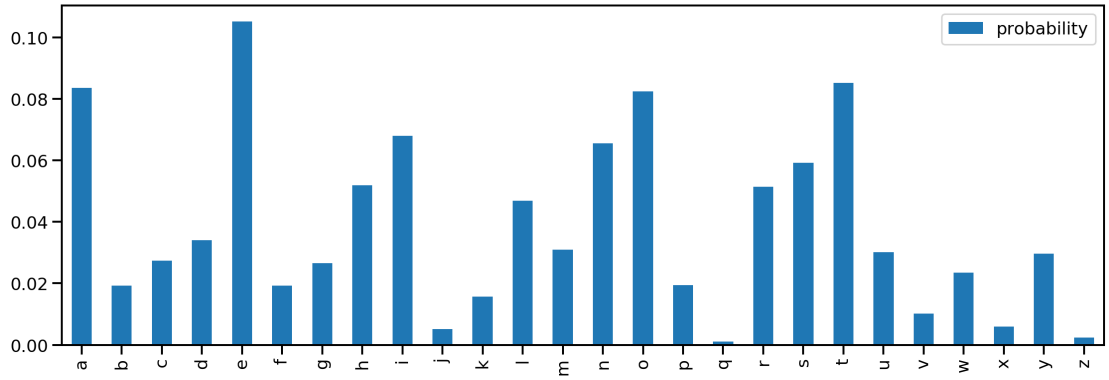


FIGURE 32. Multinomial distribution over lower-case characters.

In fact, if we sort them by their likelihood, we can see that they rapidly decline. This is another example of Zipf's Law, and can be modeled with a Zeta distribution, which we will not go into here.

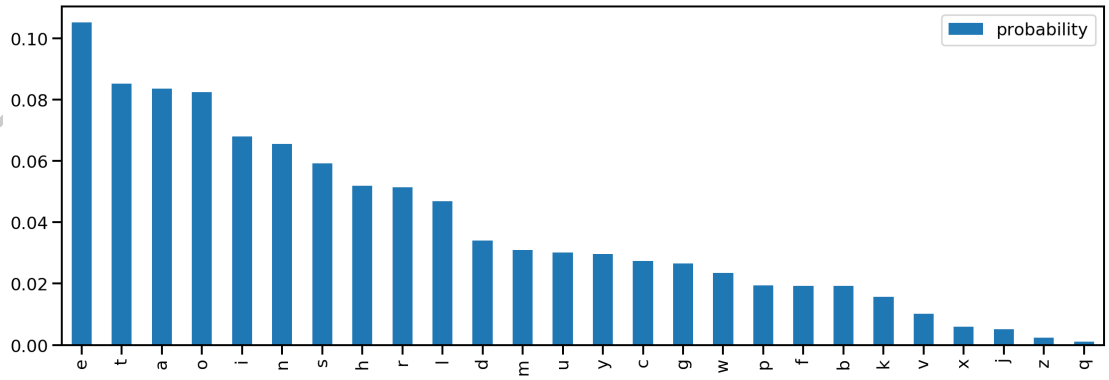


FIGURE 33. Sorted multinomial distribution over lower-case characters.

let distribu-

1.3.4. *Dirichlet distribtution.* A **Dirichlet distribution** is a distribution over multinomial distributions, and has only one parameter, called α. We can imagine a Dirichlet distribution as a generator function that gives us multinomial distributions over k outcomes for each document. The parameter α controls how peaked or uniform the multinomial distributions are: if α is close to 0, the distributions are very peaked, i.e., one outcome will have (almost) all the probability mass. The larger α gets, the more uniform the distributions become. Figure 34 shows the effect over 20 draws from a Dirichlet over 3 outcomes with different values for α.

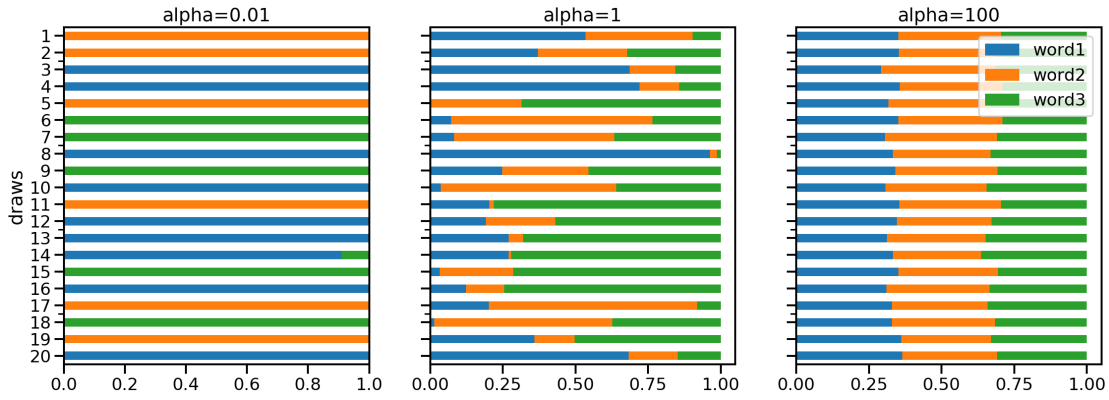


FIGURE 34. 20 multinomial distributions over three outcomes drawn from Dirichlets differing in α

2. English Stopwords

a, about, above, across, after, afterwards, again, against, all, almost, alone, along, already, also, although, always, am, among, amongst, amount, an, and, another, any, anyhow, anyone, anything, anyway, anywhere, are, around, as, at, back, be, became, because, become, becomes, becoming, been, before, beforehand, behind, being, below, beside, besides, between, beyond, both, bottom, but, by, ca, call, can, cannot, could, did, do, does, doing, done, down, due, during, each, eight, either, eleven, else, elsewhere, empty, enough, even, ever, every, everyone, everything, everywhere, except, few, fifteen, fifty, first, five, for, former, formerly, forty, four, from, front, full, further, get, give, go, had, has, have, he, hence, her, here, hereafter, hereby, herein, hereupon, hers, herself, him, himself, his, how, however, hundred, i, if, in, indeed, into, is, it, its, itself, just, keep, last, latter, latterly, least, less, made, make, many, may, me, meanwhile, might, mine, more, moreover, most, mostly, move, much, must, my, myself, name, namely, neither, never, nevertheless, next, nine, no, nobody, none, noone, nor, not, nothing, now, nowhere, of, off, often, on, once, one, only, onto, or, other, others, otherwise, our, ours, ourselves, out, over, own, part, per, perhaps, please, put, quite, rather, re, really, regarding, same, say, see, seem, seemed, seeming, seems, serious, several, she, should, show, side, since, six, sixty, so, some, somehow, someone, something, sometime, sometimes, somewhere, still, such, take, ten, than, that, the, their, them, themselves, then, thence, there, thereafter, thereby, therefore, therein, thereupon, these, they, third, this, those, though, three, through, throughout, thru, thus, to, together, too, top, toward, towards, twelve, twenty, two, under, unless, until, up, upon, us, used, using, various, very, via, was, we, well, were, what, whatever, when, whence, whenever, where, whereafter, whereas, whereby, wherein, whereupon, wherever, whether, which, while, whither, who, whoever, whole, whom, whose, why, will, with, within, without, would, yet, you, your, yours, yourself, yourselves